ELETTRONICA INFORMATICA NUOVE TECNOLOGIE

ASSEMBLER PER 68000

G. Kane D. Hawkins L. Leventhal

ASSEMBLER PER 68000

G. Kane D. Hawkins L. Leventhal

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

INTRODUZIONE ALL'EDIZIONE ITALIANA

L'MC68000, uno dei più recenti microprocessori della Motorola, è stato realizzato in modo da abbinare doti di particolare potenza ad una grande flessibilità, grazie alla disponibilità di ben 16 registri a 32 bit ed alla possibilità di indirizzare direttamente fino a 16 megabyte di memoria.

Questo volume consente anche ai lettori italiani di disporre di un testo che, oltre a fornire una completa introduzione al linguaggio assembly, descrive il set di istruzioni e le tecniche di programmazione dell'MC68000. Uno strumento indispensabile per tutti coloro, sia appassionati sia programmatori professionisti, che vogliono conoscere a fondo questo microprocessore, utilizzato su molti nuovi microcomputer. Un gran numero di esempi applicativi rende la lettura particolarmente interessante e stimolante.

RINGRAZIAMENTI

Desideriamo ringraziare la Motorola Microsystems, Mesa, Arizona ed, in particolare, Jeff Lavell, il loro ex-General Manager, per aver fornito le apparecchiature e le informazioni necessarie per scrivere questo libro.

Un grazie particolare a Mr. Hans Kalldall per il suo eccellente lavoro nel collaudo di tutti gli esempi di programmazione e per aver suggerito numerose correzioni e modifiche, che hanno notevolmente migliorato questo volume.

Doug Hawkins desidera esprimere la sua più profonda gratitudine alla moglie Peggy per l'aiuto, la pazienza e la comprensione che ha dimostrato durante la realizzazione di questo progetto.

GLI AUTORI

Gerry Kane, membro dello staff tecnico della Osborne/Mac Graw-Hill, è co-autore di numerosi volumi della famosa collana, *An Introduction to Microcomputers*. Recentemente, ha realizzato i volumi *CRT Controller Handbook e 68000 Microprocessor Handbook*, entrambi appartenenti alla nuova collana degli Osborne Handbook. Ha ricevuto il suo grado di B.S. dalla Accademia della Guardia Costiera degli Stati Uniti.

Doug Hawkins e' vice presidente del settore tecnico della Phoenix Digital Corporation, Phoenix, Arizona, ed è responsabile della progettazione e realizzazione di sistemi basati su microprocessori per il controllo di impianti di distribuzione ed il controllo di processo. In precedenza, Mr. Hawkins ha lavorato per la Motorola Microsystems, la principale produttrice dell'MC68000, come manager dei Sistemi di Linguaggio. Ha conseguito il suo BSEE presso la Michigan State University, mentre l'MSEE e l'MBA presso l'Arizona State University.

Lance Leventhal è uno dei soci della Emulative Systems Company, Inc., una società di consulenza, specializzata in microprocessori e microprogrammazione. Tiene conferenze sui microprocessori per conto della IEEE; è autore di dieci libri e di oltre sessanta articoli sui microprocessori e collabora regolarmente a pubblicazioni come *Simulation* e *Microcomputing*. Ricopre anche l'incarico di redattore tecnico della "Society for Computer Simulation", oltre a collaborare alla rivista *Digital Design*.

Il Dr. Leventhal ha scritto i primi volumi di questa collana ed ha iniziato a lavorare sulla nuova collana Assembly Language Subroutines. Ha ricevuto il suo B.A. dalla Washington University a St. Louis ed i suoi M.S. e Ph.D. presso l'University of California a San Diego. È membro della SCS, della ACM, della IEEE e della IEEE Computer Society.

INDICE

ESEMPI DI PROGRAMMAZIONE

SEZIONE I

I CONCETTI FONDAMENTALI

Questo libro descrive la programmazione in linguaggio assembly. Si presuppone che abbiate letto *An Introduction to Microcomputers: Volume 1 - Basic Concepts* (Berkeley: Osborne/Mac Graw-Hill, 1980) ed, in particolare, si fa riferimento ai Capitoli 6 e 7 di questo testo. Non ci soffermeremo sulle caratteristiche generali dei computer e dei microcomputer, sui metodi di indirizzamento o i set di istruzioni; a tale proposito vi rimandiamo, ancora una volta, a *An Introduction to Microcomputers: Volume 1*

I capitoli di questa prima parte espongono i concetti fondamentali del linguaggio assembly, in generale, e di quello dell'MC68000, in particolare. Il Capitolo 1 prende in esame le finalità di questo linguaggio, confrontandolo con i linguaggi di alto livello. Il Capitolo 2 tratta degli assemblatori e, in breve, dei caricatori. Il Capitolo 3 esamina l'architettura del microprocessore MC68000, lo confronta con i processori dello stesso tipo, analizzando, inoltre, le caratteristiche principali degli assemblatori Motorola destinati all'MC68000.

COME È STATO STAMPATO QUESTO LIBRO

Questo libro contiene parti in neretto e parti in caratteri chiari. Nei brani in chiaro vengono sviluppate le informazioni contenute nella precedente sezione in neretto. In questo modo il lettore può trascurare quegli argomenti con i quali ha già una certa familiarità, tralasciando le parti in chiaro. Nei casi in cui è opportuno, invece, un maggiore approfondimento, consigliamo di leggere sia il materiale in neretto, che quello in chiaro.

CAPITOLO 1

INTRODUZIONE ALLA PROGRAMMAZIONE IN LINGUAGGIO ASSEMBLY

Un programma è, in definitiva, una serie di numeri e, quindi, ha ben poco significato per un essere umano. In questo capitolo saranno esaminati i linguaggi simili a quello umano, con i quali è possibile scrivere un programma destinato ad un elaboratore. Si parlerà anche del modo in cui viene utilizzato il linguaggio assembly, che costituisce l'argomento del libro, analizzando i motivi che ne giustificano l'impiego.

IL SIGNIFICATO DELLE ISTRUZIONI

Definizione di "set di istruzioni"

Il set di istruzioni di un microprocessore è l'insieme degli input binari che provocano l'esecuzione di determinate azioni, durante un ciclo di istruzioni. Un set di istruzioni è per un microprocessore quello che una tabella di funzione rappresenta per un dispositivo logico, come una porta, un addizionatore o un registro di shift. Naturalmente, le azioni che un microprocessore esegue in risposta alle istruzioni ricevute sono molto più complesse di quelle che i dispositivi logici compiono in risposta ai loro input.

Istruzioni binarie

Un'istruzione è una sequenza di cifre binarie, che deve essere disponibile al momento giusto, agli ingressi dei dati del microprocessore, in modo da poter essere interpretata come un'istruzione. Ad esempio, quando il microprocessore MC68000 riceve la sequenza binaria di 16 bit 1101001100000000 come input, durante l'operazione di prelievo di un'istruzione (fase di fetch), la interpreta nel modo seguente:

"Aggiungi i contenuti del Registro Dati D0 al Registro Dati D1"

Analogamente, la sequenza
00010000001110100000000011111111 significa:

"Muovi 11111111 nel Registro Dati D0."

Il microprocessore (al pari di ogni altro computer) è in grado di riconoscere, come dati o istruzioni, solamente delle sequenze binarie; non riconosce, invece, nè dei caratteri, nè dei numeri ottali, decimali o esadecimali.

CARATTERISTICHE DI UN PROGRAMMA

Un programma è costituito da una serie di istruzioni che inducono l'elaboratore a svolgere una particolare funzione.

In realtà, il programma di un computer non comprende solo delle istruzioni; contiene anche dati e indirizzi di memoria, necessari al microprocessore per svolgere il compito definito dalle istruzioni. Chiaramente, se il microprocessore deve eseguire un'addizione, ha bisogno di due numeri da sommare e di una locazione dove mettere il risultato. Il programma deve stabilire, oltre al tipo di operazione da eseguire, anche dove reperire i dati e la destinazione del risultato.

Tutti i microprocessori eseguono le istruzioni in modo sequenziale, a meno che una di esse non cambi l'ordine di esecuzione o non arresti il microprocessore. Questo significa che il processore va a prelevare ciascuna istruzione dal successivo indirizzo di memoria, a meno che l'istruzione che sta eseguendo non gli indichi di fare altrimenti.

In ultima analisi, ogni programma è un insieme di numeri binari. Ad esempio, questo è un programma per l'MC68000, che somma i contenuti delle locazioni di memoria 6000_{16} e 6002_{16} e pone il risultato nella locazione di memoria 6004_{16}:

```
0011000000111000
0110000000000000
1101000001111000
0110000000000010
0011000111000000
0110000000000100
```

Si tratta di un programma in linguaggio macchina o programma oggetto, che, se introdotto nella memoria di un computer dotato di un microprocessore MC68000, potrebbe essere immediatamente eseguito.

IL PROBLEMA DELLA PROGRAMMAZIONE BINARIA

Ci sono molte difficoltà legate alla realizzazione di programmi oggetto o programmi binari in linguaggio macchina. Eccone alcune:

- È difficile comprendere e correggere i programmi. (Dopo alcune ore che si osservano, i numeri binari sembrano tutti uguali.).

- Introdurre il programma in un elaboratore è un'operazione che richiede molto tempo, in quanto, per ogni bit, è necessario spostare un interruttore sul pannello frontale ed, inoltre, bisogna caricare in memoria una parola per volta.
- I programmi non descrivono la funzione, che deve compiere l'elaboratore, in modo comprensibile ad un essere umano.
- Scrivere i programmi richiede molto tempo e molta fatica.
- Il programmatore commette spesso degli errori di distrazione, estremamente difficili da individuare e da correggere.

Ad esempio, **la seguente versione del programma di addizione contiene un solo bit errato. Provate a trovarlo:**

```
0011000000111000
0110000000000000
1100000001111000
0110000000000010
0011000111000000
0110000000000100
```

Al contrario dei computer, gli esseri umani non hanno molta dimestichezza con i numeri binari. I programmi binari sono sempre troppo lunghi, estenuanti, in grado solo di confondere e privi di significato. Alla fine un programmatore può riuscire a ricordare alcuni codici binari, ma è uno sforzo degno, senz'altro, di miglior causa.

L'USO DEGLI OTTALI E DEGLI ESADECIMALI

Modalità di esecuzione delle istruzioni

Possiamo migliorare, ma solo di poco, la situazione se scriviamo le istruzioni servendoci dei numeri ottali o esadecimali, anzichè di quelli binari. In questo libro useremo i numeri esadecimali, perchè sono più corti e, inoltre, rappresentano lo standard per l'industria dei microprocessori. La Tabella 1.1 fornisce le cifre esadecimali con i loro equivalenti binari. **Il programma dell'MC68000, che somma due numeri, diventerà:**

```
3038
6000
D078
6002
31C0
6004
```

Nella peggiore delle ipotesi, la versione esadecimale è più breve e non è così difficile verificarne l'esattezza.

Tabella 1.1 - Tabella di Conversione Esadecimale

Cifra Esadecimale	Equivalente Binario	Equivalente Decimale
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

È più semplice individuare eventuali errori in una sequenza di cifre esadecimali. La versione errata del programma di addizione in forma esadecimale diventerà:

```
3038
6000
C078
6002
31C0
6004
```

L'errore è molto più evidente.

Che cosa ce ne facciamo di questo programma esadecimale? Il microprocessore capisce solamente codici di istruzione binari. Se il vostro pannello frontale dispone di una tastiera esadecimale invece dei bit switch, (interruttori attraverso ognuno dei quali si specifica una cifra binaria) è possibile digitare direttamente in memoria il programma esadecimale: la logica della tastiera provvederà a tradurre le cifre esadecimali in numeri binari. Ma cosa accade se il pannello frontale ha solo dei bit switch? Il programmatore può convertire da solo le cifre esadecimali in quelle binarie, ma è un compito ripetitivo e noioso. Coloro, che ci provano, incorrono in ogni sorta di piccoli errori, come copiare la riga sbagliata, dimenticare un bit o invertire un bit o una cifra. Inoltre, una volta che il nostro programma è stato convertito, bisogna ancora metterlo in memoria mediante gli interruttori del pannello frontale.

Il Caricatore Esadecimale

Queste mansioni ripetitive ed estenuanti sono, tuttavia, un compito adatto ad un elaboratore, che non si stanca mai, non si annoia e

non fa, quasi mai, degli errori. **La soluzione consiste nello scrivere un programma che accetti i numeri esadecimali e, dopo averli convertiti in forma binaria, li metta in memoria. In realtà, un programma di questo tipo è disponibile sulla maggior parte dei microcomputer e viene chiamato "caricatore esadecimale".**

Il caricatore esadecimale è un programma come un altro e, come tale, occupa dello spazio di memoria. In alcuni sistemi, risiede in memoria il tempo sufficiente per caricare un altro programma; in altri, occupa una zona riservata della memoria di sola lettura (ROM). È probabile che il vostro microcomputer non abbia dei bit switch sul suo pannello frontale; può darsi che non abbia nemmeno un pannello frontale. Questo rispecchia le intenzioni del progettista, per il quale la programmazione binaria non solo è terribilmente noiosa, ma anche del tutto inutile. Nel vostro sistema, il caricatore esadecimale può far parte di un programma più ampio chiamato monitor, che mette a disposizione anche degli strumenti per la correzione e l'analisi dei programmi.

Un caricatore esadecimale non risolve, certamente, tutti i problemi di programmazione. Anche la versione esadecimale di un programma è difficile da leggere e da comprendere; ad esempio, non si riesce a distinguere le operazioni dai dati o dagli indirizzi ed il listato non fornisce nessuna indicazione su quello che è in grado di fare.il programma Cosa significa 3038 o 31C0? Quella di memorizzare un foglio pieno di codici non è una prospettiva molto allettante. Inoltre, i codici saranno completamente differenti per i vari microprocessori ed il programma richiederà un'ampia documentazione.

MNEMONICI DEI CODICI DI ISTRUZIONE

Un evidente miglioramento, in fase di programmazione, si ottiene assegnando un nome a ciascuna istruzione. Il nome di un codice d'istruzione si chiama mnemonico o "jogger" di memoria. Il mnemonico d'istruzione deve descrivere, con il minor numero possibile di caratteri, che cosa fa un'istruzione.

Inventare dei Mnemonici

In pratica, tutti i costruttori forniscono un set di mnemonici per le istruzioni di un microprocessore (anche loro non riescono a ricordare i codici esadecimali). **Non è necessario attenersi scrupolosamente ai mnemonici del costruttore**; non sono oggetti sacri. **Tuttavia, essi rappresentano lo standard per un dato microprocessore e, quindi, sono comprensibili a tutti gli utenti.** Sono gli stessi codici d'istruzione che ritroverete nei manuali, nelle schede, nei libri, negli articoli e nei programmi. Il problema nella scelta dei mnemonici è che non sempre si riesce ad assegnare alle istruzioni dei nomi "ovvi". Alcuni nomi lo sono (ad esempio, ADD, AND, OR), altri sono, chiaramen-

te, delle contrazioni (come SUB per sottrazione, XOR per l'OR esclusivo), mentre altri ancora non sono nè l'una nè l'altra cosa. Ne risultano certi mnemonici, come WMP, PCHL o, addirittura, SOB. La maggior parte dei costruttori ricorre a dei nomi in parte motivabili, in parte privi di ogni logica. Tuttavia, gli utenti, che inventano dei propri mnemonici, difficilmente riescono a fare di meglio.

Mnemonici Standard

È stato proposto un set standard di mnemonici per il linguaggio assembly[1]. Non sappiamo ancora se verrà adottato su larga scala, ma dovrebbe servire almeno come base di partenza per confrontare i diversi set d'istruzioni e scegliere i mnemonici per i futuri processori.

Insieme coi mnemonici d'istruzione, il costruttore, di solito, assegna dei nomi ai registri della CPU. Come accade per i nomi delle istruzioni, alcuni nomi di registri sono ovvi (A per Accumulatore), mentre altri possono avere solo un significato storico. Anche in questo caso seguiremo le indicazioni del costruttore in modo da favorire una standardizzazione.

Un Programma in Linguaggio Assembly

Se ci serviamo dei mnemonici standard, definiti dalla Motorola per le istruzioni ed i registri dell'MC68000, il nostro programma di addizione per l'MC68000 diventerà:

```
MOVE  $6000,D0
ADD   $6002,D0
MOVE  D0,$6004
```

Il programma è ancora lungi dall'esser chiaro, ma almeno alcune sue parti sono comprensibili. ADD rappresenta un notevole miglioramento rispetto a D078. I mnemonici MOVE suggeriscono realmente lo spostamento di dati in un registro o in una locazione di memoria. Adesso possiamo distinguere le parti del programma che indicano delle operazioni da quelle che rappresentano degli indirizzi. **Un programma come questo è un programma in linguaggio assembly.**

L'ASSEMBLATORE

Come facciamo ad introdurre un programma in linguaggio assembly in un computer? È necessario convertirlo in forma binaria o esadecimale. **Possiamo effettuare la conversione manualmente,** istruzione per istruzione e, in tal caso, si parlerà di assemblaggio manuale.

Assemblaggio Manuale

La tabella seguente illustra l'assemblaggio manuale del programma di addizione:

Mnem. d'istruz	Reg./Loc. di memoria	Equiv. Esad.
MOVE	$6000,D0	30386000
ADD	$6002,D0	D0786002
MOVE	D0,$6004	31C06004

Come la conversione da esadecimale a binario, anche l'assemblaggio manuale è un compito arduo, privo di interesse e soggetto a numerosi piccoli errori. Trascrivere la linea sbagliata, invertire delle cifre, tralasciare delle istruzioni o leggere erroneamente dei codici sono solo alcuni degli errori in cui è possibile incorrere. La maggioranza dei microprocessori rendono il compito ancora più complesso, avendo delle istruzioni di lunghezza diversa. Alcune istruzioni sono racchiuse in una word (parola), altre ne richiedono due o tre. Alcune istruzioni vogliono i dati nella seconda e nella terza parola; altre richiedono indirizzi di memoria, numeri dei registri o chissà cosa.

Assemblaggio Automatico

L'assemblaggio è un altro compito ingrato che possiamo delegare all'elaboratore, il quale non commette mai degli errori nel tradurre i codici e sa sempre quante word e quale formato richiede ciascuna istruzione. Il programma che assolve questa funzione viene detto "assemblatore". L'assemblatore traduce il programma "sorgente" scritto dall'utente usando i mnemonici, in un programma in linguaggio macchina, o programma "oggetto", che può essere eseguito dal computer. L'input dell'assemblatore è costituito da un programma sorgente e l'output è un programma oggetto.

Sebbene l'assemblatore sia un programma, proprio come lo è il caricatore esadecimale, è più difficile da realizzare, occupa più memoria, richiede un maggior numero di periferiche ed ha un tempo di esecuzione molto più lungo. Mentre molti utenti realizzano i loro caricatori, sono pochi coloro che si cimentano nella realizzazione di un assemblatore.

Gli assemblatori hanno delle regole ben precise, che devono essere rispettate. Esse riguardano l'uso di certi indicatori (come gli spazi, le virgole, il punto e virgola o i due punti) in posizioni opportune, una corretta ortografia, un adeguato controllo delle informazioni e, in certi casi, il corretto posizionamento dei nomi e dei numeri. Si tratta, in genere, di regole semplici e facili da imparare.

Ulteriori Caratteristiche degli Assemblatori

I primi assemblatori si limitavano a tradurre i mnemonici delle istruzioni e dei registri nei loro equivalenti binari. La maggior parte degli assemblatori attuali, invece, svolge funzioni supplementari, quali:

Funzioni principale dell'assemblatore

- Permettere all'utente di assegnare dei nomi alle locazioni di memoria, ai dispositivi di ingresso e uscita ed anche a sequenze di istruzioni
- Convertire in forma binaria dati o indirizzi forniti in forma diversa (per esempio decimale o esadecimale) e convertire i caratteri nei loro codici binari ASCII o EBCDIC
- Effettuare alcune operazioni aritmetiche, durante la fase di assemblaggio
- Indicare al caricatore in quale zona di memoria mettere determinate parti del programma o i dati
- Consentire all'utente di definire aree di memoria per immagazzinare temporaneamente dei dati e di mettere dati costanti nella zona riservata alla memoria di programma
- Fornire le informazioni necessarie per includere programmi standard, contenuti in una libreria, o programmi scritti in altre occasioni
- Permettere all'utente di controllare il formato del listato del programma e i dispositivi di input/output impiegati

La Scelta di un Assemblatore

Criteri di scelta

Naturalmente, tutte queste caratteristiche comportano costi e memoria supplementari. Generalmente i microcomputer hanno degli assemblatori molto più semplici rispetto ai computer più grandi, ma le dimensioni degli assemblatori tendono sempre ad aumentare. Spesso vi capiterà di trovarvi di fronte ad una vasta gamma di assemblatori. Ciò che deve guidare nella scelta non è la presenza di caratteristiche fuori del comune, ma piuttosto la maggiore o minore funzionalità durante la normale utilizzazione.

SVANTAGGI DEL LINGUAGGIO ASSEMBLY

L'assemblatore, come il caricatore esadecimale, non risolve tutti i problemi della programmazione. Resta sempre il profondo divario fra il set d'istruzioni di un microcomputer e le funzioni che questo deve svolgere. Le istruzioni tendono a fare cose come sommare i contenuti di due registri, shiftare di un bit il contenuto dell'Accumulatore o porre un nuovo valore nel Contatore di Programma. D'altra parte, un normale utente vuole che l'elaboratore stampi un numero, attenda un particolare comando da una telescrivente, e risponda in modo adeguato, oppure provveda ad attivare un relè al momento opportuno. Un programmatore in liguaggio assembly deve tradurre questi compiti in una sequenza di semplici istruzioni comprensibili al microprocessore. La traduzione può essere un lavoro lungo e difficile.

Inoltre, **se programmate in linguaggio assembly, dovete avere una conoscenza dettagliata del microcomputer che state utilizzando.** Dovete conoscere i registri e le istruzioni, sapere con esattezza come queste agiscono sui vari registri, quali sono i possibili tipi di indirizzamento e un'infinità di altre cose. Nessuna di queste è attinente alla funzione che, in definitiva, il microcomputer dovrà svolgere.

Mancanza di Portabilità

Oltre a questo, i programmi in linguaggio assembly non sono portabili. Ogni microcomputer ha un proprio linguaggio assembly, che rispecchia la sua architettura. Un programma in linguaggio assembly destinato all'MC68000 non girerà su un microprocesore 6809, un 8080 o uno Z8000. Ad esempio, il programma di addizione per lo Z8000 bisognerebbe scriverlo in questa forma:

```
LD   R0,%6000
ADD  R0.%6002
LD   %6004,R0
```

La mancanza di portabilità non significa, soltanto, che non potrete far girare un vostro programma in linguaggio assembly su un microcomputer diverso, ma anche che non vi sarà possibile usare un programma, che non sia stato scritto specificatamente per il tipo di microcomputer, che state utilizzando. Questo è uno svantaggio, particolarmente per i microcomputer a 16 bit, come l'MC68000, dal momento che si tratta di dispositivi nuovi e sono pochi i programmi disponibili scritti nel loro linguaggio assembly.

I LINGUAGGI AD ALTO LIVELLO

La soluzione alle molte difficoltà legate alla programmazione in linguaggio assembly è quella di usare i linguaggi ad "alto livello" o "orientati alle procedure". Si tratta di linguaggi che consentono di descrivere una certa funzione in una forma connessa con le caratteristiche di quel particolare problema, senza tener conto del computer utilizzato. In un linguaggio ad alto livello, ogni istruzione svolge una funzione riconoscibile, che, di solito, corrisponde a molte istruzioni in linguaggio assembly. Un programma chiamato compilatore ha il compito di tradurre un programma sorgente, scritto in un linguaggio ad alto livello, in codice oggetto, cioè in una serie di istruzioni in linguaggio macchina.

UN LINGUAGGIO AD ALTO LIVELLO: FORTRAN

Linguaggi ad alto livello per microcomputer

Esistono molti linguaggi ad alto livello per i più diversi tipi di applicazioni. **Se, ad esempio, si desidera esprimere ciò che deve fare il computer in notazione algebrica, allora ci si può servire del FORTRAN (Formula Translation Language), il più vecchio ed il più utilizzato fra i linguaggi ad alto livello.** Volendo sommare due numeri, basterà dire al computer:

```
SUM = NUM1 + NUM2
```

È un programma molto più semplice (e molto più breve) di uno analogo scritto in linguaggio macchina o in linguaggio assembly. Ai linguaggi ad alto livello appartengono il COBOL (destinato ad applicazioni commerciali), il PL/1 (una combinazione di FORTRAN e COBOL), l'APL (progettato per scrivere programmi molto compatti), il BASIC (diffuso sui microcomputer più piccoli), il C (un linguaggio per la programmazione di sistemi sviluppato presso i laboratori della Bell Telephone) ed il Pascal (progettato per la programmazione strutturata).

VANTAGGI DEI LINGUAGGI AD ALTO LIVELLO

È evidente che i linguaggi ad alto livello rendono più semplice e rapida la stesura dei programmi. Secondo una valutazione piuttosto diffusa, scrivere un programma in un linguaggio ad alto livello richiede ad un programmatore un tempo dieci volte inferiore rispetto ad un programma in linguaggio assembly[2-4]. Questo solo per scrivere il programma; senza tener conto delle fasi di definizione del problema, di progettazione, di correzione, di collaudo e documentazione: risulteranno tutte più semplici e rapide. Un programma scritto in un linguaggio ad alto livello, in parte, si documenta da solo. Anche chi non conosce il FORTRAN, sarà probabilmente in grado di comprendere il significato della linea di programma che vi abbiamo appena mostrato.

Indipendenza dalla Macchina

I linguaggi ad alto livello risolvono molti altri problemi relativi alla programmazione in linguaggio assembly. Un linguaggio ad alto livello ha una propria sintassi (generalmente definita da uno standard nazionale o internazionale) e non fa rifermento al set d'istruzioni, ai registri o ad altre caratteristiche di un determinato computer. Di

tutti questi dettagli si occupa il compilatore. I programmatori possono concentrarsi sui loro compiti, senza dover conoscere nei dettagli l'architettura della CPU: al limite, non hanno bisogno di sapere neanche per quale computer stanno programmando.

Portabilità

I programmi scritti in un linguaggio ad alto livello sono "portabili", almeno in teoria. Potranno essere utilizzati su qualsiasi computer che disponga di un compilatore standard per quel linguaggio.

Allo stesso tempo, tutti i precedenti programmi scritti per altri computer in un linguaggio ad alto livello sono disponibili per essere impiegati su un computer di nuova produzione. Nel caso di linguaggi molto diffusi, come il FORTRAN o il BASIC, questo significa poter disporre di migliaia di programmi.

SVANTAGGI DEI LINGUAGGI AD ALTO LIVELLO

Se tutte le cose positive dette a proposito dei linguaggi ad alto livello sono vere (cioè, che la stesura dei programmi è molto più rapida e, inoltre, si ha la garanzia di una completa portabilità), perchè perdere tempo con il linguaggio assembly? Chi ha voglia di preoccuparsi dei registri, dei codici d' istruzione, dei mnemonici e di tutto il resto! Come spesso accade, anche nell'uso dei linguaggi di alto livello esistono degli svantaggi.

Sintassi

Un problema abbastanza scontato, con qualunque linguaggio ad alto livello è quello di **doverne imparare le "regole" e la "sintassi", come, del resto, accade con il linguaggio assembly.** Un linguaggio ad alto livello ha un insieme di regole piuttosto complesso. Impiegherete parecchio tempo solo per imparare a scrivere un programma sintatticamente corretto (ed anche allora probabilmente non otterrete i risultati preventivati). Un linguaggio ad alto livello è come una lingua straniera. Chi ha una certa predisposizione, si abituerà alle regole e sarà in grado di realizzare dei programmi, che saranno accettati dal compilatore. Comunque, imparare le regole e riuscire a far compilare un programma non significa, necessariamente, aver raggiunto lo scopo che vi eravate prefissi.
Ecco, ad esempio, alcune regole per i compilatori FORTRAN:

- Le label devono essere numeri posti nelle prime cinque colonne
- Le istruzioni devono cominciare nella settima colonna
- Le variabili intere devono iniziare con le lettere I, J, K, L, M o N

Costo dei Compilatori

Un altro problema evidente è la necessità di disporre di un compilatore per tradurre in linguaggio macchina un programma scritto in un linguaggio ad alto livello. I compilatori sono piuttosto costosi ed utilizzano una notevole quantità di memoria. Mentre la gran parte degli assemblatori occupa da 2K a 16Kbyte di memoria, i compilatori ne occupano da 4K a 64K. Ne deriva che gli svantaggi legati all'impiego di un compilatore non sono del tutto trascurabili.

La Scelta del Linguaggio Adatto

Inoltre, **solo alcuni compilatori rendono più semplice la stesura di un determinato programma.** Il FORTRAN, ad esempio, ben si adatta a problemi che possono essere espressi mediante formule algebriche. Se, tuttavia, dovete gestire un terminale video, fare l'edit di una stringa di caratteri o controllare un sistema d'allarme, usare il FORTRAN non sarà di grande utilità. In realtà, risulterà più scomodo e difficile formulare la soluzione in FORTRAN che in linguaggio assembly. Il rimedio consiste, naturalmente, nell' utilizzare un linguaggio ad alto livello più adatto. Esistono dei linguaggi progettati specificatamente per assolvere queste funzioni: sono i linguaggi sviluppati appositamente per la realizzazione di sistemi. Il loro impiego, tuttavia, è molto limitato, come del resto il grado di standardizzazione.

Inefficienza

I linguaggi ad alto livello non consentono di ottenere programmi in linguaggio macchina molto efficienti. La ragione fondamentale va ritrovata nel fatto che la compilazione è un processo automatico, che deve ricorrere a dei compromessi per far fronte ad una vasta gamma di situazioni. Il compilatore funziona come un traduttore computerizzato: in certi casi le parole sono giuste, ma la struttura della frase risulta un pò goffa. Un semplice compilatore non può sapere quando una variabile non viene più utilizzata e può quindi essere abbandonata, quando è meglio usare un registro invece di una locazione di memoria o quando delle variabili hanno dei rapporti semplici. Il programmatore esperto può avvalersi di scorciatoie, in modo da abbreviare il tempo di esecuzione o ridurre la memoria utilizzata. Alcuni compilatori (noti come compilatori ottimizzanti) riescono a farlo da soli, ma occupano molta più memoria rispetto ai normali compilatori.

SOMMARIO DEI VANTAGGI E DEGLI SVANTAGGI

Vantaggi dei Linguaggi ad Alto Livello:

- Facilità nell'apprenderli (ed insegnarli ad altri)
- Una descrizione più efficiente delle diverse funzioni
- Una più rapida stesura dei programmi
- Maggiore facilità di documentazione
- Una sintassi standard
- Indipendenza dalla struttura di un particolare computer
- Portabilità
- Disponibilità di una libreria di programmi

Svantaggi dei Linguaggi ad Alto Livello:

- Regole speciali
- Necessità di una grossa struttura hardware e software
- I linguaggi più diffusi sono adatti soprattutto a impieghi di tipo scientifico o commerciale
- Programmi poco efficienti
- Difficoltà ad ottimizzare il codice per soddisfare eventuali necessità di tempo e di memoria
- Impossibilità di sfruttare al meglio le particolari caratteristiche di un computer

LINGUAGGI AD ALTO LIVELLO PER I MICROPROCESSORI

Soprattutto coloro che utilizzano dei microprocessori incontreranno molte difficoltà nell'impiego di linguaggi ad alto livello. Fra queste ci sono le seguenti:

- **Esistono pochi linguaggi ad alto livello destinati a dei microprocessori.** Questo è vero soprattutto per i nuovi microprocessori e per quelli relativamente poco diffusi o rivolti ad applicazioni di controllo.
- **Sono pochi i linguaggi standard disponibili.**
- **I compilatori richiedono, in genere, una notevole quantità di memoria o, addirittura, un altro computer completamente diverso.**
- Molte delle applicazioni dei microprocessori si adattano poco ai linguaggi ad alto livello.
- Molti linguaggi riservati ai microprocessori non permettono di ottenere un programma oggetto. Questo significa che il programma viene tradotto ed eseguito linea per linea (si parla di linguag-

gio interpretato e non compilato). **In altri casi è necessario disporre di un particolare supporto software ('run-time package'), per poter eseguire il programma.** In entrambi i casi, i programmi risultano molto lenti e richiedono una grande quantità di memoria. Il BASIC ed il PASCAL, i linguaggi ad alto livello più diffusi, utilizzano, generalmente, l'uno o l'altro di questi metodi.

- **I costi di memoria sono spesso critici nelle applicazioni che utilizzano dei microprocessori.**

La scarsa disponibilità di linguaggi ad alto livello destinati ai microcomputer è da attribuire al fatto che i microprocessori sono prodotti relativamente recenti ed hanno visto la luce nell'industria dei semiconduttori e non in quella dei computer. Fra i linguaggi disponibili troviamo il BASIC[5], il Pascal[6,7], il FORTRAN, il C[8] e i linguaggi tipo PL/I, come il PL/M[9].

Molti tra i linguaggi ad alto livello destinati ai microcomputer non sono conformi a degli standard riconosciuti, per cui gli utenti di microprocessori non possono attendersi di ottenere grossi risultati in termini di portabilità, di poter accedere a biblioteche di programmi o utilizzare esperienze e programmi precedenti. Restano, comunque, i vantaggi derivanti da una maggiore facilità di programmazione e di documentazione e dal fatto che non è necessaria una conoscenza dettagliata dell'architettura di un computer.

Costi relativi ai Linguaggi ad Alto Livello

I costi legati all'impiego di un linguaggio ad alto livello con i microprocessori sono considerevoli. Fino a poco tempo fa, i microprocessori risultavano più indicati per applicazioni di controllo e per applicazioni interattive lente, piuttosto che per la manipolazione di caratteri e l'analisi di linguaggio necessaria in fase di compilazione. Perciò, molti compilatori non sono in grado di girare su sistemi dotati di microprocessore ma richiedono elaboratori molto più grandi: si tratta cioè di compilatori incrociati (cross-compilers) e non di auto-compilatori (self-compilers). Quindi, oltre ad affrontare la spesa di un computer più grande, è necessario anche, una volta compilato il programma, trasferirlo sul microcomputer per il quale è stato realizzato.

Naturalmente, sono disponibili anche dei compilatori in grado di girare sullo stesso microcomputer per il quale forniscono il codice oggetto. Purtroppo, richiedono spesso una grande disponibilità di memoria e particolari supporti hardware e software.

Applicazioni non adatte ai Linguaggi ad Alto Livello

Inoltre, i linguaggi ad alto livello non si rivelano adeguati per certe applicazioni dei microprocessori. Gran parte dei linguaggi più

comuni è stata progettata per risolvere problemi scientifici o per gestire l'elaborazione di dati commerciali su larga scala. Molte applicazioni dei microprocessori non rientrano in questo ambito, ma riguardano, piuttosto, l'invio di dati e di informazioni di controllo a dispositivi di uscita e la ricezione di dati e di informazioni di stato di dispositivi di ingresso. Spesso queste informazioni sono rappresentate da poche cifre binarie, con un significato ben preciso, legato all'hardware utilizzato Se provate a scrivere un programma di controllo in un linguaggio ad alto livello, vi sentirete come uno che cerca di mangiare la minestra con i bastoncini. Negli impieghi riguardanti apparecchiature di collaudo, terminali, sistemi di navigazione, elaborazione di segnali e impieghi commerciali, i linguaggi ad alto livello si rivelano molto più efficaci che nel campo della strumentazione, delle comunicazioni, delle periferiche e delle applicazioni automobilistiche.

Applicazioni per i diversi Livelli di Linguaggio

Le applicazioni più adatte ai linguaggi ad alto livello sono quelle che comportano una grande disponibilità di memoria. Se, come accade nel caso di un videogame oppure quando si deve controllare una valvola o un qualsiasi altro dispositivo, anche il costo di un solo chip di memoria è importante, allora un uso inefficiente della memoria, tipico dei linguaggi ad alto livello, diventa intollerabile. Se, d'altra parte, come in un terminale o in un'apparecchiatura di test, il sistema ha, in ogni caso, molte migliaia di byte a disposizione, un eventuale spreco di memoria finisce per essere trascurabile. È chiaro che anche le dimensioni ed il volume del prodotto sono elementi di cui bisogna tener conto. Un programma molto esteso depone a favore dell'impiego di un linguaggio ad alto livello. D'altra parte, nelle applicazioni di grande diffusione i costi relativi allo sviluppo del software non sono così importanti come il costo della memoria.

QUALE LIVELLO USARE?

La scelta del livello di linguaggio da impiegare dipende dal tipo di applicazione. Consideriamo alcuni degli elementi che devono orientare nella scelta:

Linguaggio Macchina:

- In pratica nessuno programma in linguaggio macchina, dato che ciò comporta uno spreco di tempo e una considerevole difficoltà in fase di documentazione. Inoltre, bisogna considerare che un assemblatore costa molto poco e riduce notevolmente il tempo di programmazione.

Linguaggio Assembly:
- Programmi brevi o, comunque, non eccessivamente lunghi
- Applicazioni in cui il costo della memoria rappresenta un aspetto decisivo
- Applicazioni di controllo in tempo reale
- Elaborazioni di una quantità limitata di dati
- Applicazioni di grande volume
- Applicazioni che comportamo operazioni di input/output e funzioni di controllo, piuttosto che di calcolo

Linguaggi ad Alto Livello
- Programmi lunghi
- Applicazioni di limitata diffusione
- Applicazioni in cui la quantità di memoria disponibile è già molto grande
- Applicazioni che comportano prevalentemente dei calcoli, anzichè funzioni di controllo o di input/output
- Compatibilità con applicazioni simili, che utilizzano elaboratori più grossi
- Disponibilità di programmi in linguaggi ad alto livello, che possono essere utilizzati per quella applicazione
- Programmi che si presume dovranno subire molti cambiamenti

Altre Considerazioni

Vanno considerati anche altri aspetti, quali la disponibilità, in fase di sviluppo, di un grosso computer, il grado di esperienza con determinati linguaggi e la compatibilità con altre applicazioni.

Se è l'hardware che incide maggiormente sul costo della vostra applicazione oppure è la velocità l'elemento critico, bisogna optare a favore del linguaggio assembly, tenendo sempre presente che, in cambio di minori costi di memoria e di una maggiore velocità di esecuzione, saranno richiesti tempi più lunghi per lo sviluppo del software.

Naturalmente nessuno, fatta eccezione per qualche purista, troverà qualcosa da obiettare se utilizzerete sia i linguaggi ad alto livello che l'assembly. Potete scrivere inizialmente un programma in un linguaggio ad alto livello e, successivamente, inserire alcune parti in linguaggio assembly.[10,11] Tuttavia, molti preferiscono non farlo, per evitare la confusione che ne potrebbe derivare in fase di correzione, di collaudo e di documentazione.

TENDENZE PER IL FUTURO

Riteniamo che in futuro verranno utilizzati quasi esclusivamente i linguaggi ad alto livello, per i seguenti motivi:

- I programmi diventano sempre più estesi per l'aggiunta di sempre nuove caratteristiche
- Il costo delle componenti hardware e della memoria diminuisce
- Il software ed i programmatori stanno diventando più costosi
- Sono disponibili chip di memoria di dimensioni sempre maggiori e ad un costo "per bit" nettamente inferiore, per cui è meno probabile un eventuale risparmio sui chip
- Si stanno sviluppando linguaggi ad alto livello più adatti ed efficienti
- Aumenta la disponibilità dei compilatori
- Si va verso una maggiore standardizzazione dei linguaggi ad alto livello

La programmazione dei microprocessori in linguaggio assembly non sarà un'arte in via di estinzione, più di quanto lo sia per i grandi elaboratori. Ma programmi più lunghi, memorie più economiche e programmatori più costosi faranno sì che il software finisca per incidere, in modo prevalente, sul costo finale della gran parte delle applicazioni. C'è, dunque, una tendenza a preferire l'impiego dei linguaggi ad alto livello.

Perchè questo Libro ?

Se il futuro sembra favorire i linguaggi ad alto livello, perchè un libro sulla programmazione in linguaggio assembly? I motivi sono questi:

1. Gran parte degli utenti di microcomputer nell'ambito dell'industria programmano in linguaggio assembly (almeno i due terzi, secondo una recente inchiesta).
2. Molti utenti di microcomputer continuerranno a programmare in linguaggio assembly, perchè hanno bisogno del controllo accurato che esso è in grado di fornire.
3. Non c'è ancora una grande disponibilità di linguaggi ad alto livello, che, inoltre, non sono sufficientemente standardizzati.
4. **Tutti, o quasi, i programmatori di microcomputer scoprono, prima o poi, di aver bisogno di una certa conoscenza del linguaggio assembly,** soprattutto per correggere i programmi, scrivere delle routine di input/output, velocizzare o accorciare sezioni critiche di programmi scritti in linguaggi ad alto livello, sfruttare o modificare certe funzioni del sistema operativo e riuscire a capire i programmi scritti da altri.
5. La conoscenza del linguaggio assembly può essere utile anche nella valutazione dei linguaggi ad alto livello

La parte restante di questo libro verterà esclusivamente sugli assemblatori e la programmazione in linguaggio assembly. Vogliamo, tuttavia, che i lettori siano a conoscenza del fatto che il linguaggio assembly non è la sola alternativa. È opportuno prestare atten-

zione a eventuali nuovi sviluppi, che possano ridurre, in modo significativo, i costi di programmazione, se tali costi rappresentano un aspetto importante.

BIBLIOGRAFIA

1. W. P. Fischer, "Microprocessor Assembly Language Draft Standard", *Computer*, December 1979, pp. 96-109.
2. M.H. Halstead, *Elements of Software Science,* American Elsevier, New York, 1977.
3. L.H. Putnam and A. Fitzsimmons, "Estimating Software Costs", *Datamation,* September 1979, pp. 189-98.
4. M. Phister, Jr., *Data Processing Technology and Economics,* Santa Monica Publishing Co., Santa Monica Calif., 1976. Disponibile anche presso la Digital Press, Educational Services Digital Equipment Corp., Bedford, Mass.
5. Albrecht Finkel, and Brown, *BASIC for Home Computers,* Wiley, New York 1978.
6. G.M. Schneider et al., *An Introduction to Programming and Problem Solving with PASCAL,* Wiley, New York, 1978.
7. K.L. Browles, *Microcomputer Problem Solving Using PASCAL,* Springer-Verlag, New York, 1977.
8. B.W. Kernighan and D.M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., 1978.
9. D.D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications,* Addison-Wesley, Reading, Mass., 1978.
10. P. Caudill, "Using Assembly Coding to Optimize High-Level Language Programs," *Electronics*, Frebruary 1, 1979, pp. 121-24.
11. D.B. Wecker et al., "High Level Design Language Develops Low Level Microprocessor-Indipendent Software", *Computer Design,* June 1979, pp. 140-49.

CAPITOLO 2

ASSEMBLATORI

Questo capitolo prende in esame gli assemblatori, iniziando dalle caratteristiche comuni alla maggior parte di essi e proseguendo con la descrizione di funzioni più complesse, quali le macro e l'assemblaggio condizionato. Chi lo desideri, può saltare, per il momento, questo capitolo e ritornarvi quando avrà una maggiore familiarità con l'argomento.

CARATTERISTICHE DEGLI ASSEMBLATORI

Come abbiamo ricordato in precedenza, gli assemblatori attuali non si limitano soltanto a tradurre i mnemonici del linguaggio assembly in codici binari. Ma, prima di descriverne le ulteriori caratteristiche, vogliamo soffermarci sulle modalità con cui avviene la traduzione dei mnemonici. Infine, spiegheremo l'utilizzazione degli assemblatori.

I campi del linguaggio assembly

Codice operativo

Le istruzioni del linguaggio assembly (o "statement") sono divise in un certo numero di "campi", com'è indicato nella Tabella 2-1.

Il campo del codice operativo è il solo che non può essere mai vuoto; contiene sempre un mnemonico d'istruzione o una direttiva per l'assemblatore, detta anche "pseudo-istruzione", "pseudo-operazione" o "pseudo-op".

Operando

L'operando o campo indirizzi può contenere un indirizzo o un dato o, in alcuni casi, essere vuoto.

Tabella 2.1 - I Campi di un'Istruzione in Linguaggio Assembly

Campo Etichetta	Campo Codice Operativo o Mnemonico	Campo Operandi o Indirizzi	Campo del Commento
VALUE1:	DC.W	$201E	Primo valore
VALUE2:	DC.W	$0774	Secondo valore
RESULT:	DS.W	1	16 Bit per salvare il risultato
.			
.			
.			
START	MOVE	VALUE1,D0	Prendi il primo valore
	ADD	VALUE2,D0	Somma il secondo valore al primo
	MOVE	D0,RESULT	Memorizza il risultato della somma
NEXT:	?		Istruzione successiva

Commento a label

I campi riservati al commento ed alla label (etichetta) sono opzionali. Un programmatore assegna una label ad uno statement o aggiunge un commento, solo per motivi di convenienza personale: ad esempio per rendere più comprensibile la struttura del programma e facilitarne l'uso.

Naturalmente, l'assemblatore deve disporre di un mezzo per indicare la fine di un campo e l'inizio di quello successivo. Molti assemblatori esigono che un campo abbia inizio in una colonna ben precisa. Si parla, in questo caso, di "formato rigido". Tuttavia, i formati rigidi causano parecchi problemi (soprattutto quando il mezzo di input è rappresentato dal nastro di carta) e non sono graditi ai programmatori. L'alternativa è il "formato libero", in cui i campi possono avere inizio in un qualsiasi punto della riga.

Delimitatori

Se l'assemblatore non può usare la posizione sulla riga per indicare la separazione dei campi, deve servirsi di qualcos'altro. **La maggior parte degli assemblatori utilizza un simbolo speciale o "delimitatore" all'inizio o alla fine di ogni campo.** Il delimitatore più comune è il carattere di spazio. Anche le virgole, i punti, il punto e virgola, i due punti, i trattini, i punti interrogativi e altri caratteri che resterebbero altrimenti inutilizzati in un programma in linguaggio assembly, possono avere la funzione di delimitatori. La Tabella 2-2 elenca i delimitatori standard per gli assemblatori dell'MC68000.

È necessario fare attenzione con i delimitatori. Alcuni assemblatori sono molto esigenti a proposito degli spazi in più o in meno o riguardo alla presenza di delimitatori nei commenti o nelle label. Un buon assemblatore sarà in grado di risolvere da solo questi piccoli problemi, ma, purtroppo, molti assemblatori non sono scritti bene. Quello che vi raccomandiamo è di evitare, se possibile, problemi inutili. Le regole seguenti vi saranno di aiuto:

Regole per l'uso dei delimitatori

- Non usare spazi in più, in particolare dopo le virgole che separano gli operandi.
- Non usare caratteri delimitatori nei nomi o nelle etichette.
- Utilizzare i delimitatori standard, anche se il vostro assemblatore ne consente altri. Questo aumenterà le probabilità che il vostro programma sia compatibile anche con altri assemblatori.

Tabella 2.2 - Delimitatori Standard degli Assemblatori dell'MC68000

"spazio"	Tra la label ed il codice operativo, tra il codice operativo e l'indirizzo e prima di un eventuale commento
virgola	Fra gli operandi del campo indirizzi
asterisco	Prima di una riga interamente destinata al commento

Label

Quello della label (o etichetta) è il primo campo di un'istruzione in linguaggio assembly; può anche essere vuoto. Se è presente una label, l'assemblatore le assegna il valore dell'indirizzo in cui viene caricato il primo byte del codice oggetto relativo a quella istruzione. La stessa label può, successivamente, essere usata come un indirizzo o un dato nel campo indirizzi di un'altra istruzione. L'assemblatore, durante la creazione del programma oggetto, provvederà a sostituirla con il valore che le è stato assegnato.

L'assemblatore della Motorola si serve di due delimitatori diversi per indicare la fine del campo riservato alle etichette. Se una label comincia all'inizio di una riga, come abbiamo visto nel paragrafo precedente, allora è uno spazio che deve indicarne la fine. Tuttavia, questo assemblatore permette che una label abbia inizio in un punto qualunque della riga, nel qual caso sono necessari i due punti (:) come delimitatore del campo.

Utilizzazione di una label

L'uso delle etichette è molto frequente soprattutto nelle istruzioni di Jump, Branch o TRAP. Queste istruzioni pongono un nuovo valore nel contatore di programma, alterando così la normale sequenza di esecuzione. JMP 15016 significa "metti il valore 15016 nel contatore di programma". La successiva istruzione, che viene eseguita, è quella contenuta nella locazione di memoria 15016. L'istruzione JMP START significa "metti il valore assegnato alla label START nel contatore di programma". L'istruzione successiva sarà quella che si trova all'indirizzo indicato dalla label START. La Tabella 2-3 mostra un esempio.

Perchè usare una label? Questi sono alcuni dei motivi:

- Una label serve ad individuare una locazione del programma ed a ricordarla meglio.
- Una label può facilmente essere spostata, se necessario, per cambiare o correggere un programma. Quando il programma viene riassemblato, l'assemblatore provvederà a cambiare automaticamente tutte le istruzioni che fanno uso di quella etichetta.
- Un assemblatore o un caricatore possono rilocare l'intero programma, aggiungendo una costante (costante di rilocazione) a ciascun indirizzo, in cui compare una label. In questo modo, diventa possibile spostare il programma, per permettere l'inserimento di altre routine o semplicemente per riordinare la memoria.
- È più facile usare un programma come programma di libreria; cioè, chiunque altro può servirsene come subroutine di un altro programma completamente diverso.
- Non sarà necessario calcolare gli indirizzi di memoria, compito estremamente arduo con quei microprocessori che hanno istruzioni di lunghezza variabile.

Si dovrebbe assegnare una etichetta ad ogni istruzione, cui, in seguito, si voglia far riferimento.

Formato di una label

L'altro problema riguarda il modo in cui scegliere la label. L'assemblatore pone spesso alcune restrizioni riguardanti il numero dei caratteri (in genere 5 o 6), il carattere iniziale (spesso deve trattarsi di una lettera) e i caratteri successivi (devono essere lettere, numeri oppure dei caratteri speciali). Al di là di queste limitazioni, la scelta dipende da voi.

Tabella 2.3 - Assegnazione ed Uso di una Label

Programma in Linguaggio Assembly

```
START   MOVE VALUE 1,D0
        .
        .
        .  (PROGRAMMA PRINCIPALE)
        .
        .
        .
        JMP START
```

Quando viene eseguita la versione in linguaggio macchina di questo programma, l'istruzione JMP START pone l'indirizzo dell'istruzione indicata dalla label START nel contatore di programma e sarà questa la prossima istruzione ad essere eseguita.

Da parte nostra, consigliamo di **adottare per le label dei nomi che suggeriscono la loro funzione,** cioè delle label mnemoniche. Ad esempio, ADDW per una routine che aggiunge una parola in una somma, SRCHETX per una che ricerca il carattere ETX oppure NKEYS per una locazione della memoria dati che contiene il numero di tasti premuti. Etichette che hanno un certo significato si ricordano meglio e servono anche a documentare un progamma. Alcuni programmatori adottano un formato standard per le label, iniziando, ad esempio, con L000. Sono usate in modo sequenziale (si possono saltare alcuni numeri per permettere inserimenti successivi), ma non servono certo a documentare meglio il programma.

Le regole seguenti vi saranno di aiuto nella scelta delle label, evitandovi molti problemi:

- Non usare label uguali ai codici operativi o ad altri mnemonici. La gran parte degli assemblatori non lo consente; altri sì, ma si finisce per fare confusione.
- Non usare label di lunghezza superiore a quella prevista dall'assemblatore. Gli assemblatori hanno regole diverse e spesso ignorano alcuni caratteri alla fine di una label troppo lunga.
- Evitare i caratteri speciali (non alfabetici e non numerici) e le lettere minuscole. Alcuni assemblatori non li permettono, altri ne riconoscono solo alcuni. Il metodo più semplice è di servirsi solo delle lettere maiuscole e dei numeri.
- Iniziare ogni etichetta con una lettera. Solo così sarete sicuri che verranno accettate da tutti gli assemblatori.

- Non usare label che possono essere confuse tra di loro. Evitare le lettere I, O e Z e i numeri 0, 1 e 2. e label come XXXX e XXXXX. Non ha senso sfidare il destino e la Legge di Murphy ("se c'è la possibilità che una cosa vada storta, andrà sicuramente storta").
- Quando non siete sicuri se una label è legittima, non usatela. Non ricaverete nessun beneficio, scoprendo, a vostre spese, ciò che l'assemblatore è in grado di riconoscere .

Queste sono soltanto raccomandazioni, non delle regole fisse. Non siete obbligati a rispettarle, ma non rimproverate noi, se poi vi trovate a perder tempo su dei problemi, che avreste potuto evitare.

Codici operativi dell'assemblatore (mnemonici)

Una delle funzioni principali di un assemblatore è la traduzione dei codici operativi mnemonici nei loro equivalenti binari. L'assemblatore svolge questo compito utilizzando una tabella proprio come nell'assemblaggio manuale.

Un assemblatore non si deve, tuttavia, limitare alla traduzione dei codici operativi. **Il suo compito è anche quello di stabilire quanti e quali operandi richiede una determinata istruzione.** Questo può rivelarsi piuttosto complesso: alcune istruzioni (come Stop) non hanno operandi, altre (come l'istruzione Jump) ne hanno uno solo, mentre altre ancora (come un'operazione di trasferimento tra registri o uno shift multiplo) ne richiedono due. Alcune istruzioni consentono anche delle alternative; ad esempio, alcuni computer hanno istruzioni (come Shift o Clear) che possono applicarsi ad un registro della CPU oppure ad una locazione di memoria. Non ci soffermiamo a descrivere in che modo l'assemblatore riesce a fare una distinzione; ci limitiamo soltanto a prenderne nota.

Direttive all'assemblatore

Utilità delle direttive

Alcune istruzioni del linguaggio assembly non vengono tradotte direttamente in istruzioni in linguaggio macchina. Queste istruzioni sono direttive destinate all'assemblatore; servono a stabilire l'indirizzo iniziale del programma, a definire dei simboli, a riservare certe zone della RAM per la memorizzazione dei dati, a mettere in memoria delle tabelle o delle costanti, a fare riferimento a label contenute in altri programmi ed a svolgere piccole funzioni di riordino ("housekeeping").

Come si usa una direttiva

Per usare queste direttive o pseudo-operazioni, un programmatore deve porre il mnemonico della direttiva nel campo del codice operativo e, se una direttiva lo richiede, anche un indirizzo o un dato nel campo indirizzi.

Le direttive più usate sono:
DATA
EQUATE o DEFINE
ORIGIN
RESERVE

Direttive di collegamento fra programmi diversi sono:
ENTRY
EXTERNAL

Gli assemblatori usano nomi diversi per queste operazioni, ma le funzioni sono le stesse. Le direttive di housekeeping sono:
END
LIST
NAME
PAGE
SPACE
TITLE
PUNCH

Nella parte successiva descriveremo brevemente queste pseudo-operazioni, anche se le loro funzioni sono abbastanza ovvie.

La Direttiva DATA

La direttiva DATA o DEFINE CONSTANT consente al programmatore di introdurre delle costanti nella memoria di programma. Questi dati possono comprendere:

- Tabelle di consultazione
- Tabelle per la conversione di codici
- Messaggi
- Sequenze di sincronizzazione
- Valori soglia
- Nomi
- Coefficienti di equazioni
- Comandi
- Fattori di conversione
- Valori di confronto
- Tempi o frequenze caratteristici
- Indirizzi di subroutine
- Identificazioni di tasti
- Sequenze di collaudo
- Sequenze per la generazione di caratteri
- Tabelle di registrazione
- Modelli standard
- Maschere
- Tabelle per le transizioni di stato

La direttiva DATA considera i dati come componenti permanenti del programma.

Il formato di una direttiva DATA è molto semplice. Un'istruzione del tipo:

```
DZCON    DATA    12
```

metterà il numero 12 nella successiva locazione di memoria disponibile e assegnerà a quella locazione il nome DZCON. Ogni direttiva DATA ha, di solito, una label, a meno che non appartenga ad una serie. Il dato e la label possono assumere uno qualsiasi dei formati consentiti dall'assemblatore.

Molti assemblatori permettono di usare delle direttive DATA più complesse, capaci di gestire, contemporaneamente, una notevole quantità di dati. Ad esempio:

```
EMESS    DATA    'ERROR'
SQRS     DATA    1,4,9,16,25
```

Con una sola istruzione si possono riempire molti byte della memoria di programma, pur dovendo rispettare i limiti dovuti alla lunghezza della riga o alle restrizioni poste dal particolare tipo di assemblatore impiegato. Naturalmente, è possibile aggirare queste limitazioni, servendoci di più direttive DATA, una dopo l'altra:

```
MESSG      DATA    'ORA È IL'
           DATA    'MOMENTO PER TUTTI GLI'
           DATA    'UOMINI DI BUONA VOLONTÀ'
           DATA    'DI ACCORRERE IN'
           DATA    'AIUTO DEL LORO'
           DATA    'PAESE'
```

Gli assemblatori destinati ai microprocessori consentono molto spesso alcune variazioni rispetto alla direttiva DATA standard. DEFINE BYTE o FORM CONSTANT BYTE gestiscono valori ad 8 bit; DEFINE WORD o FORM CONSTANT WORD valori ed indirizzi a 16 bit. Sono disponibili anche altre direttive speciali per i dati codificati in forma di caratteri.

La Direttiva EQUATE (o DEFINE)

La direttiva EQUATE consente al programmatore di assegnare dei nomi ai dati ed agli indirizzi. Questa pseudo-operazione è quasi sempre indicata con il mnemonico EQU o =. I nomi possono riferirsi ad indirizzi di periferiche, a dati numerici, alla locazione iniziale del programma, a indirizzi costanti, ecc.

La direttiva EQUATE assegna il valore numerico, presente nel campo dell'operando, alla relativa label. Ecco due esempi:

```
TTY     EQU    5
LAST    EQU    5000
```

La gran parte degli assemblatori consente di definire una label in termini di un'altra, come ad esempio:

```
LAST    EQU    FINAL
ST1     EQU    START+1
```

L'etichetta usata in un campo operando deve, naturalmente, essere stata definita in precedenza. Spesso, il campo operando contiene espressioni più complesse, come vedremo meglio in seguito. Assegnazioni con doppio nome (due nomi per lo stesso dato o per lo stesso indirizzo) sono utili per fondere programmi che usano nomi diversi per indicare una stessa variabile (o un'ortografia diversa per quella che dovrebbe essere la stessa parola!).

Effetto della direttiva EQUATE

Si noti che una direttiva EQU non induce l'assemblatore a mettere un dato in memoria, ma semplicemente ad inserire un nuovo nome in una tabella (detta "tabella dei simboli" o "symbol table"), che deve sempre essere tenuta aggiornata.

Quando si deve usare un nome? Tutte le volte che usate un parametro che pensate di modificare successivamente o al quale volete assegnare un significato particolare, al di là del semplice valore numerico. Di solito si assegnano dei nomi alle costanti di tempo, agli indirizzi delle periferiche, alle maschere, ai fattori di conversione, ecc. Nomi come DELAY, TTY, KBD, KROW o OPEN non solo rendono più semplice cambiare un parametro, ma contribuiscono anche a documentare meglio un programma. Solitamente vengono assegnati dei nomi anche a delle locazioni di memoria, utilizzate per degli scopi particolari, come memorizzare dei dati, indicare l'inizio del programma o conservare temporaneamente dei dati.

Quali nomi usare? Le regole da seguire sono più o meno le stesse usate per le etichette, ad eccezione del fatto che in questo caso è ancor più importante servirsi di nomi che abbiano un certo significato. Perchè non chiamare una telescrivente TTY, anzichè X15, o indicare un bit di ritardo con BTIME o BTDLY, invece di WW, e il numero del tasto "GO" con GOKEY, invece di HORSE? Benchè possano sembrare avvertenze piuttosto scontate, sono molti i programmatori che se ne dimenticano.

Dove mettere le direttive EQUATE? Il posto migliore è all'inizio del programma, accompagnandole con degli opportuni commenti, sul tipo di INDIRIZZI DI I/O, DEPOSITO TEMPORANEO, COSTANTI DI TEMPO o LOCAZIONI DI PROGRAMMA. In questo modo diventa più facile individuare delle definizioni e poter-

le eventualmente modificare. Per di più, un altro utente sarà agevolato dal trovare tutte le definizioni raggruppate in uno stesso punto. Chiaramente, questa abitudine contribuisce a rendere più chiaro un programma ed a facilitarne l'uso.

Le definizioni impiegate soltanto in una particolare subroutine dovrebbero comparire all'inizio della subroutine stessa.

La Direttiva ORIGIN

La direttiva ORIGIN (di solito abbreviata in ORG) consente al programmatore di specificare le locazioni di memoria, in cui devono trovarsi i programmi, le subroutine e i dati. Programmi e dati possono occupare zone diverse della memoria, a seconda della configurazione del sistema. Le routine di inizializzazione, quelle destinate a servire gli interrupt o altre che assolvono compiti speciali possono essere disseminate nella memoria disponibile, a seconda delle particolari necessità.

Effetto della direttiva ORIGIN

L'assemblatore tiene aggiornato un contatore di locazione (paragonabile al contatore di programma del computer), che contiene la locazione di memoria corrispondente al dato o all'istruzione che, in quel momento, viene processato. Una direttiva ORG induce l'assemblatore a mettere un nuovo valore nel contatore di locazione, un po' quello che fa l'istruzione Jump, che costringe la CPU a modificare il contenuto del contatore di programma. L'output dell'assemblatore non deve contenere soltanto una serie di istruzioni e di dati, ma deve anche indicare al caricatore in quale parte della memoria metterli.

I programmi destinati ad un microprocessore contengono spesso più di uno statement ORG, allo scopo di indicare le locazioni seguenti:

- Indirizzo di Reset (o Startup)
- Indirizzi per il servizio degli inerrupt
- Indirizzi di Trap (interrupt di tipo software)
- Aree della RAM destinate a conservare dei dati
- Stack
- Programma principale
- Subroutine
- Indirizzi di memoria per i dispositivi di I/O o per funzioni speciali

Altro uso della direttiva ORIGIN

Si possono, inoltre, utilizzare altri statement ORIGIN allo scopo di riservare dello spazio per eventuali inserimenti successivi, mettere dei dati o delle tabelle in una determinata area di memoria o assegnare delle zone della RAM a funzioni di buffer dati. Nei microcomputer, programmi e dati possono occupare zone completamente diverse della memoria, in modo da rendere più semplice l'hardware.

Tipici statement ORIGIN sono:

```
ORG    RESET
ORG    1000
ORG    INT3
```

Se il programmatore non mette uno statement ORG all'inizio del programma, molti assemblatori presuppongono un'origine zero La convenienza è minima, ma vi raccomandiamo di includere uno statement ORG per evitare confusione.

La Direttiva RESERVE

La direttiva RESERVE o DEFINE STORAGE consente al programmatore di riservare delle aree di memoria per scopi diversi, quali tabelle, deposito temporaneo di dati, indirizzi indiretti, uno Stack, ecc.

Mediante la direttiva RESERVE è possibile assegnare un nome ad una zona della memoria e indicare il numero delle locazioni che devono farne parte. Ecco alcuni esempi:

```
NOKEY    RESERVE    1
TEMP     RESERVE    50
VOLTG    RESERVE    80
BUFR     RESREVE    100
```

Si può usare la direttiva RESERVE per riservare un certo numero di locazioni della memoria di programma o di quella dati; tuttavia, solitamente essa è impiegata nella memoria dati.

Effetto della direttiva RESERVE

In realtà, la direttiva RESERVE si limita ad incrementare il contatore di locazione del valore presente nel campo dell'operando, senza che l'assemblatore produca un vero e proprio codice oggetto.

Caratteristiche di questa direttiva:

1. **L'etichetta corrispondente alla direttiva RESERVE viene assegnata all'indirizzo della prima delle locazioni riservate.** Per esempio, la psuedo-operazione:

   ```
   TEMP    RESERVE    20
   ```

   riserva 20 parole di memoria e assegna il nome TEMP all'indirizzo del primo byte
2. **È necessario specificare il numero dello locazioni, non essendo previsto un valore di default.**
3. **Nessun dato viene messo nelle locazioni riservate.** Qalunque dato si trovi in queste locazioni, resterà immutato.

Alcuni assemblatori permettono di definire i valori iniziali da inserire nelle locazioni riservate. Vi consigliamo di non sfruttare questa possibilità, in quanto questo presuppone che il programma (insieme

con i valori iniziali) sia caricato da un dispositvo esterno (ad esempio, nastro di carta o floppy disk), ogni volta che deve essere eseguito. I programmi destinati ai microproceesori, d'altra parte, risiedono spesso nella memoria di sola lettura (ROM) ed hanno inizio nel momento in cui si accende la macchina. La memoria dati (indicata spesso come memoria ad accesso casuale o 'RAM'), in questo caso, non conserva il suo contenuto, nè viene ricaricata con nuovi valori. Quindi, il programma deve contenere sempre una sequenza di istruzioni per l'inizializzazione della RAM: in questo modo siamo sicuri che l'inizializzazione avviene tutte le volte che il programma viene eseguito e non solo quando viene caricato.

Direttive di Link

Riferimenti Esterni

Capita spesso di impiegare, in un programma o in una subroutine, dei nomi definiti in una fase di assemblaggio diversa: sono i cosiddetti "riferimenti esterni". Questo rende necessario un particolare programma, chiamato "linker", che ha il compito di sostituire i valori corrispondenti e di stabilire se alcuni nomi sono stati definiti più volte oppure non sono stati ancora definiti.

La direttiva EXTERNAL, di solito abbreviata in EXT o XREF, indica che il nome ad essa associato viene definito altrove.

La direttiva ENTRY, abbreviata in ENT o XDEF, indica che un nome, definito in quel programma, può essere utilizzato altrove.

Il modo esatto in cui sono implementate le **direttive di linking** varia notevolmente da un assemblatore all'altro. Non ritorneremo su queste direttive, ma vi accorgerete che **sono molto importanti nelle applicazioni pratiche.**

Direttive per il Controllo dell'Output

Esistono varie direttive che modificano il funzionamento dell'assemblatore ed il listato del programma, senza avere alcun effetto sul programma oggetto. Fra le più comuni ricordiamo:

- END, che indica la fine del programma sorgente in linguaggio assembly.
- LIST, che indica all'Assemblatore di stampare il programma sorgente. Sono possibili alcune varianti come NO LIST o LIST SYMBOL TABLE.
- NAME o TITLE, che permettono di stampare un titolo all'inizio di ogni pagina del listato.
- PAGE o SPACE, che causano un salto all pagina o alla riga successiva, migliorando l'aspetto del listato e facilitandone la lettura.
- PUNCH, che trasferisce il codice oggetto successivo al perforatore di nastro. Questa pseudo-operazione rappresenta, in alcuni casi, l'opzione di default e, quindi, non è necessario inserirla.

Quando Usare le Label

Spesso alcuni utenti si domandano se e quando è possibile assegnare una label ad una direttiva dell'assemblatore. Queste sono le nostre raccomandazioni:

- **Tutte le direttive EQUATE devono avere delle label;** altrimenti sono inutili dal momento che il loro scopo è di definire le relative label.
- **Le direttive DATA e RESERVE hanno di solito una label,** che identifica la prima locazione di memoria usata o riservata.
- **Le altre direttive non devono avere necessariamente una label.** Alcuni assemblatori lo consentono, ma vi raccomandiamo di non usarne perchè non esiste uno standard per la loro interpretazione.

OPERANDI E INDIRIZZI

La maggioranza degli assemblatori garantiscono al programmatore una notevole libertà nella descrizione dei contenuti del campo indirizzi. Va ricordato, però, che l'assemblatore prevede determinati nomi per i registri e le istruzioni e, a volte, anche per altri scopi. Descriveremo, adesso, alcuni dei metodi comunemente adottati per indicare l'operando.

Numeri Decimali

Nella maggior parte dei casi, gli assemblatori presuppongono di aver a che fare con dei numeri decimali, a meno di una diversa indicazione. Per cui:

```
ADD     100
```

significa "somma il contenuto della locazione di memoria 100_{10} al contenuto dell'Accumulatore."

Altri Sistemi Numerici

Gli assemblatori accettano, di solito, anche input in forma binaria, ottale o esadecimale. Ma **è indispensabile indicare in qualche modo il sistema numerico adottato:** ad esempio, facendo precedere o seguire il numero da un carattere o da una lettera di identificazione:

- B o % per i numeri binari

- O, @, Q o C per quelli ottali (la lettera O è da evitarsi per non confonderla con lo zero)
- H o $ per gli esadecimali (o lo standard BCD)
- D per i decimali, se non rappresentano l'opzione di default.

Spesso gli assemblatori richiedono che i numeri esadecimali inizino con una cifra (ad esempio, 0A36 invece di A36), allo scopo di distinguere fra i numeri ed i nomi o le etichette. È buona norma inserire i numeri nella forma che garantisce la maggiore chiarezza: cioè le costanti decimali in forma decimale; gli indirizzi ed i numeri in BCD in forma esadecimale; maschere o sequenze di bit in uscita in forma binaria se sono abbastanza brevi, altrimenti in forma esadecimale.

Nomi

Nel campo operando possono comparire dei nomi, che verranno trattati come i dati che rappresentano. Si ricordi, tuttavia, che **esiste una differenza fra operandi e indirizzi.** In un programma in linguaggio assembly destinato all'MC68000 la sequenza:

```
FIVE    EQU     5
        ADDI    FIVE,D0
```

somma il contenuto della locazione di memoria 5 (non necessariamente il numero 5) al contenuto del registro dati D0. Una sequenza che, invece, aggiunge proprio il numero 5 è la seguente:

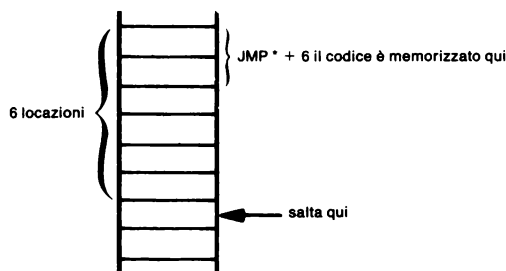
```
FIVE    EQU     5
        ADD     #FIVE,D0
```

Il simbolo # informa l'assemblatore che il numero rappresentato dal nome FIVE è il valore dell'operando FIVE stesso e non il contenuto della locazione di memoria indicata da FIVE.

Il Contatore di Locazione

È possibile utilizzare **il valore attuale del contatore di locazione, che è generealmente indicato con * o $. Questo si rivela utile soprattutto nelle istruzioni di Jump; ad esempio:**

```
JMP     *+6
```

provoca un Jump alla locazione di memoria che si trova 6 byte oltre quella contenente il primo byte dell'istruzione JMP.

Uno dei motivi che giustificano l'uso di questa tecnica è la possibilità di ridurre il numero dei simboli presenti in un programma in linguaggio assembly, soprattutto se l'assemblatore è in grado di gestire solo un numero limitato di simboli. In questo modo, diminuisce anche il tempo necessario all'assemblaggio. Si tratta, comunque, di benefici trascurabili, a meno che il programma non sia estremamente lungo e si disponga di un assemblatore primordiale.

La maggioranza dei microprocessori ha delle istruzioni formate da due o tre byte. Questo rende difficile stabilire con esattezza la distanza fra due istruzioni in linguaggio assembly. Per questo motivo, usando dei valori di spostamento relativi al contatore di locazione si commettono spesso degli errori, che si possono evitare con l'impiego delle label. In ogni caso, **è meglio non utilizzare il simbolo del contatore di locazione.**

Codici di Carattere

Quasi tutti gli assemblatori permettono di introdurre dei testi sotto forma di stringhe ASCII, racchiuse all'interno di virgolette singole o doppie. Alcuni assemblatori usano anche dei simboli per indicare l'inizio e la fine di una stringa, quali A o C. **Un numero limitato di assemblatori riconosce anche stringhe di tipo EBCDIC, lo stesso tipo adottato dalle macchine IBM.**

Raccomandiamo di usare sempre delle stringhe di caratteri per indicare un testo, al fine di migliorare la chiarezza e la leggibilità del programma.

Espressioni Aritmetiche e Logiche

Gli assemblatori consentono di combinare fra loro dei dati, rappresentati in una delle forme descritte in precedenza, mediante degli operatori logici, aritmetici o di tipo speciale. Queste combinazioni sono chiamate espressioni. Mentre tutti gli assemblatori consentono delle semplici espressioni aritmentiche come START + 1, ce ne sono alcuni che danno la possibilità di eseguire anche moltiplicazioni, divisioni, funzioni logiche, shift, ecc. Si ricordi che l'assemblatore

valuta un'espressione al momento dell'assemblaggio e se vi compare un simbolo, viene sostituito con l'indirizzo ad esso corrispondente (cioè, il contatore di locazione o il valore assegnato con la direttiva EQUATE).

Gli assemblatori si differenziano riguardo al tipo di espressioni che sono in grado di accettare e nel modo in cui le interpretano. L'uso di espressioni molto complesse rende difficile leggere un programma e capirne la struttura.

Raccomandazioni generali

A questo punto desideriamo riassumere le raccomandazioni che vi abbiamo fatto in questa parte del libro, aggiungendone qualcuna di nuova. **Come regola generale, un utente deve cercare di ottenere il massimo della chiarezza e della semplicità. Non si trae nessun vantaggio dallo sfruttare tutte le possibilità più insolite offerte da un assemblatore o dall'utilizzare espressioni particolarmente complesse. Consigliamo un approccio di questo tipo:**

- Per indicare i dati usare sempre il sistema numerico o il codice di caratteri che garantisce la maggiore chiarezza.
- Maschere e numeri BCD in forma decimale, caratteri ASCII in ottale o semplici costanti numeriche in esadecimale non servono a nulla e, quindi, non dovrebbero essere usate.
- Ricordarsi di distinguere i dati dagli indirizzi.
- Non usare valori di spostamento rispetto al contatore di locazione.
- Limitarsi ad espressioni semplici e chiare. Non fidarsi delle possibilità offerte dall'assemblatore.

ASSEMBLAGGIO CONDIZIONATO

Alcuni assemblatori permettono di inserire o escludere alcune parti del programma sorgente, a seconda di determinate condizioni esistenti al momento dell'assemblaggio: è il cosiddetto assemblaggio condizionato, che conferisce ad un assemblatore parte della flessibilità propria dei compilatori. **La maggior parte degli assemblatori destinati ad un microcomputer hanno delle capacità di assemblaggio condizionato molto limitate. Questo è un esempio tipico:**

```
IF      COND
•
•       (PROGRAMMA DA ESEGUIRE)
•
•
ENDIF
```

Se al momento dell'assemblaggio, l'espressione COND è vera, le istruzioni comprese fra IF e ENDIF (due pseudo-operazioni) vengono incluse nel programma.

Gli usi più comuni dell'assemblaggio condizionato sono:

- Includere o escludere variabili supplementari
- Inserire diagnostici o condizioni particolari, durante esecuzioni di collaudo
- Consentire dati con un diverso numero di bit.

Sfortunamente, l'assemblaggio condizionato tende ad appesantire un programma ed a renderne difficile la lettura. Conviene usarlo solo in caso di necessità.

Input dell'Assemblatore	Output dell'Assemblatore	
Programma Sorgente	**Codice Oggetto**	**Mnemonici Corrispondenti**
(Definizione di una Macro) MACRO ADDQ #1,D0 LSL.L #1,D1 BPL *-6 ENDM Fine della definizione		
(Inizio del programma principale)	E309 D140 4A41 66F8	LSL.B #1,D1 ADDX D0,D0 TST D1 BNE PARITY LOOP MAC1
LSL.B #1,D1 ADDX D0,D0 TST D1 BNE PARITY LOOP MAC1	5240 E389 6AFA	ADDQ #1,D0 LSL.L #1,D1 BPL *-6 MAC1
	80000099 00000080	BTST #0,D0 BNE NEXT CHAR
BTST #0,D0 BNE NEXT CHAR MAC1	5240 E389 6AFA	ADDQ #1,D0 LSL.L #1,D1 BPL *-6
BSET #7,-(A0)	08E80007FFFF	BSET #7,-1(A0)

Figura 2-1. Espansione di una Macro ad opera dell'Assemblatore.

MACRO

Spesso particolari sequenze di istruzioni compaiono parecchie volte all'interno di uno stesso programma sorgente. Questo può essere dovuto alla logica del programma oppure alla necessità di compensare alcune carenze del set d'istruzioni del microprocessore. Si può evitare di scrivere ripetutamente la stessa sequenza di istruzioni, ricorrendo ad una "macro"[1].

Finalità delle macroistruzioni

Le macro permettono di assegnare un nome ad una sequenza di istruzioni e di utilizzarlo, al posto di questa, nel programma sorgente. L'assemblatore provvederà a sostituire il nome della macro con la sequenza corrispondente. Le parti ombreggiate della Figura 2-1 mostrano come l'assemblatore tratta le macro presenti in un programma. Non vi preoccupate di capire cosa fa il programma ed il significato delle istruzioni; osservate solo in che modo l'assemblatore provvede ad espandere la macro MAC1.

Una macro assomiglia, in un certo senso, ad una subroutine, in quanto si tratta di un riferimento abbreviato ad una sequenza di istruzioni usata frequentemente. Non è, però, la stessa cosa. Il codice di una subroutine compare solo una volta nel programma e, durante l'esecuzione, si verifica una diramazione alla subroutine. Invece, tutte le volte che l'assemblatore incontra il nome di una macro, lo sostituisce con la relativa sequenza di istruzioni; non si ha, quindi, una diramazione in fase di esecuzione come accade con una subroutine. Il nome di una macro è una direttiva per l'assemblatore definita dall'utente e non riguarda l'esecuzione del programma, ma solo la fase di assemblaggio.

Vantaggi delle Macro:

- Programmi sorgente più brevi
- Programmi meglio documentati
- Impiego di sequenze di istruzioni già collaudate, quindi la sicurezza che quella parte del programma non contiene degli errori.
- È più facile apportare delle modifiche. Una volta cambiata la definizione della macro, l'assemblatore provvederà a sostituirla, ogni volta che ne incontra il nome.
- La possibilità di inserire nuove istruzioni nel set di base, allo scopo di ampliarlo e renderlo più chiaro.

Svantaggi delle Macro

- Il fatto che una macro è espansa ogni volta che viene usata, comporta uno spreco di memoria, dovuto alla ripetizione della stessa sequenza d'istruzioni.
- Una singola macro può produrre una grande mole di istruzioni.
- La mancanza di uno standard rende difficile comprendere la struttura di un programma.
- Difficoltà a stabilire eventuali effetti collaterali a carico dei registri o dei flag.

COMMENTI

Tutti gli assemblatori consentono di inserire dei commenti nel programma sorgente. I commenti non hanno alcun effetto sul codice

oggetto, ma servono a comprendere meglio la struttura e le finalità di un programma. Dei buoni commenti rappresentano una componente essenziale della stesura di un programma.

Parleremo dei commenti e delle tecniche di documentazione in un capitolo successivo. Per il momento, ci limitiamo ad alcune indicazioni di carattere generale:

- I commenti servono ad indicare la funzione che il programma sta svolgendo e non a spiegare il significato delle istruzioni.
  Ecco alcuni esempi: "LA TEMPERATURA HA SUPERATO IL LIMITE?", "LINE FEED ALLA TTY" oppure "ESAMINA L'INTERRUTTORE DI CARICAMENTO".
  Commenti del tipo "SOMMA 1 ALL'ACCUMULATORE", "VAI A START" o "GUARDA IL CARRY" sono del tutto inutili. Dobbiamo descrivere in che modo il programma agisce sull'intero sistema; gli effetti interni sulla CPU risultano già evidenti dai mnemonici delle istruzioni.
- Usare commenti brevi e pertinenti. Per i dettagli fate riferimento ad altre parti della documentazione.
- Commentare tutti i punti chiave.
- È inutile commentare le istruzioni standard o le sequenze che modificano contatori o puntatori. Bisogna, invece, prestare una particolare attenzione alle istruzioni che risultino poco chiare.
- Non usare abbreviazioni incomprensibili.
- Fare commenti precisi e leggibili.
- Commentare tutte le definizioni, descrivendone la funzione. Indicare tutte le tabelle e le aree destinate a contenere dei dati.
- Commentare intere sezioni del programma e non limitarsi alle singole istruzioni.
- Essere coerenti nell'uso della terminologia, anche rischiando di essere ripetitivi. Non è necessario consultare un dizionario di sinonimi.
- Mettete delle note nei punti che vi sembrano più confusi: ad esempio, "RICORDA CHE IL CARRY È STATO MESSO A 1 DALL'ULTIMA ISTRUZIONE". Se con il successivo sviluppo del programma, queste note si riveleranno inutili, potrete sempre toglierle.

Un programma ben commentato è facile da usare e vi farà risparmiare molto tempo, certamente più di quello impiegato a scrivere i commenti. Vi mostreremo come dovrebbero essere dei buoni commenti negli esempi di programmazione, che vi forniremo nei capitoli successivi, anche se qualche volta finiremo per essere prolissi, allo scopo di chiarire meglio particolari aspetti del linguaggio assembly.

TIPI DI ASSEMBLATORI

Sebbene tutti gli assemblatori svolgano le stesse funzioni, spesso sono molto diverse le modalità di realizzazione. Non sarebbe possi-

bile prendere in esame tutti i tipi di assemblatori disponibili, ma ci limiteremo a descrivere quelli più utilizzati, indicando alcune delle possibili alternative.

Cross-assembler

Un cross-assembler (lett. assemblatore incrociato) **è un assemblatore che gira su un computer diverso da quello per cui deve assemblare il programma oggetto.**

Si tratta in genere di un grosso computer con un notevole supporto software e con delle periferiche molto veloci, come un IBM 370, un Univac 1108 o un Burroughs 6700. Elaboratori di questo tipo sono impiegati per assemblare programmi destinati per lo più a dei micro, come il 6809 o l'MC68000. Molti cross-assembler sono scritti in FORTRAN o in un altro linguaggio ad alto livello, in modo da garantirne la portabilità.

Quando viene prodotto un nuovo microcomputer viene fornito anche un cross-assembler in grado di girare sui sistemi di svilupppo già esistenti. Ad esempio la Motorola fornisce un cross-assembler per l'MC68000 che gira sui sistemi di sviluppo dotati di un 6809.

Auto-assembler

Un auto-assemblatore o assemblatore residente è un assemblatore che gira sullo stesso computer per il quale assembla i programmi. Richiede una certa disponibilità di memoria e di periferiche e risulterà molto lento se paragonato ad un cross-assembler.

Macro-assembler

Un macroassemblatore è un assemblatore che permette di definire delle sequenze di istruzioni sotto forma di macro.

Micro-assembler

Un microassemblatore è un assemblatore usato per scrivere dei microprogrammi, che definiscono il set di istruzioni di un microprocessore. La microprogrammazione non ha niente a che fare con la programmazione dei microcomputer, ma concerne il funzionamento interno di un computer[2,3].

Meta-assembler

Un meta-assemblatore è un assemblatore che può gestire molti set di istruzioni diversi. L'utente deve specificare il particolare set che desidera utilizzare.

One-pass assembler

Un assemblatore ad un passaggio (one-pass assembler) è un assemblatore che analizza il programma in linguaggio assembly soltanto una volta. Un assemblatore di questo tipo deve avere un modo per assegnare un valore a quei simboli che compaiono ad un certo punto del programma, ma sono definiti solo in una parte successiva ("forward references"), come ad esempio quelle istruzioni di Jump che usano label non ancora definite.

Two-pass assembler

Un assemblatore a due passaggi (two-pass assembler) è un assemblatore che esamina il programma sorgente in linguaggio assembly due volte. La prima volta si limita a raccogliere e definire i simboli; la seconda sostituisce i riferimenti con le definizioni vere e proprie. Un assemblatore di questo tipo non ha problemi quando si tratta di gestire dei simboli non ancora definiti, ma può essere molto lento se non è disponibile una memoria di massa (come un floppy disk), in quanto l'assemblatore dovrebbe fisicamente leggere il programma due volte da un mezzo di input molto lento (come un lettore di nastro di carta per telescrivente). La maggioranza degli assemblatori destinati a dei microprocessori richiede due passaggi.

ERRORI

Gli assemblatori forniscono normalmente dei messaggi d'errore, costituiti per lo più da un codice di una sola lettera. Questi sono alcuni degli errori possibili:

- Un nome non definito (spesso a causa di un errore d'ortografia o di una dimenticanza)
- Un carattere non ammesso (come un 2 in un numero binario)
- Un formato non ammesso (un delimitatore sbagliato o un operando errato)
- Un'espressione non valida (ad esempio, due operatori di seguito l'uno all'altro)
- Un valore non consentito (di solito troppo grande)
- La mancanza di un operando
- Una duplice definizione (due valori diversi indicati con lo stesso nome)
- Una label non ammessa (come una label assegnata ad una pseudo-operazione che non può averne)
- La mancanza di una label
- Un codice operativo non previsto.

Nell'interpretarne gli errori, non va dimenticato che l'assemblatore può prendere una strada sbagliata, se trova una lettera fuori posto, uno spazio di troppo o una punteggiature scorretta. Le istruzioni successive saranno lette in modo sbagliato e si otterranno messaggi d'errore privi di senso. Bisogna sempre osservare attentamente il primo errore, in quanto è da esso che possono dipendere quelli successivi. Una certa accortezza ed il rispetto dei formati standard vi eviteranno molti errori fastidiosi.

CARICATORI

Il caricatore (loader) è un programma che ha il compito di prelevare l'output (codice oggetto) dell'assemblatore e di metterlo in memoria. Esistono molti tipi di caricatori, dai più semplici ai più complessi. Ne descriveremo solo alcuni.

Bootstrap loader

Un caricatore di tipo bootstrap (bootstrap loader) è un programma che usa le sue prime istruzioni per caricare in memoria la parte restante oppure un altro caricatore. Può trovarsi nella ROM o può essere introdotto nella memoria di un elaboratore mediante i switch del pannello frontale. In certi casi è l'assemblatore stesso che provvede a mettere un caricatore di questo tipo all'inizio del programma oggetto.

Relocating loader

Un caricatore "rilocante" carica i programmi in una parte qualsiasi della memoria. Di solito carica ogni programma nella zona di memoria immediatamente successiva a quella utilizzata dal program-

Absolute loader

Linking loader

ma precedente. L'importante è che i programmi possano essere spostati, cioè che siano rilocabili. **Un caricatore "assoluto", al contrario, metterà i programmi sempre in una stessa area di memoria.**

Un caricatore con funzioni di link (linking loader) carica programmi e subroutine assemblati separatamente, risolvendo i riferimenti incrociati, cioè le istruzioni di un programma che fanno riferimento a label definite in un altro. Può utilizzare soltanto programmi oggetto generati da un assemblatore, che permette riferimenti esterni ("external references"). È possibile anche separare le fasi di caricamento e di link, in modo che quest'ultima funzione possa essere affidata a un programma specifico (link editor).

BIBLIOGRAFIA

1. A complete monograph on macros is M. Campbell-Kelly, *An Introduction to Macros* American Elsevier, New York, 1973.
2. A. Osborne, *An Introduction to Microcomputers: Volume I - Basic Concepts,* Osborne/Mc Graw-Hill, Berkeley, Calif., 1980.
3. A.K. Agrawala and T.G. Rauscher, *Foundations of Microprogramming,* Academic Press, New York, 1976.
4. D.K. Barron *Assemblers and Loaders,* American Elsevier, New York, 1972.
5. C.W. Gear, *Computer Organization and Programming,* McGraw-Hill, New York, 1974.

CAPITOLO 3

IL LINGUAGGIO ASSEMBLY E LA STRUTTURA DELL'MC68000

Questo capitolo descrive, nei suoi tratti generali, la struttura del microprocessore MC68000 e fornisce le regole sintattiche dell'assemblatore Motorola. Le caratteristiche hardware, insieme con i segnali di uscita e le interfacce, sono descritte più ampiamente in *The 68000 Microprocessor Handbook*[1]. Da parte nostra ci limiteremo a considerare l'MC68000 dal punto di vista del programmatore in linguaggio assembly, per il quale pins e segnali non hanno nessuna importanza e non c'è molta differenza fra un minicomputer ed un microcomputer. Nei capitoli successivi parleremo dello stack dell'MC68000 e del trattamento delle Exception.

Nelle tabelle seguenti abbiamo elencato il set di istruzioni dell'MC68000, distinguendo fra istruzioni usate molto spesso (Tabella 3-1), istruzioni usate solo occasionalmente (Tabella 3-2) e istruzioni usate pochissime volte (Tabella 3-3). I programmatori esperti non troveranno importante una simile distinzione e potranno anche non condividerla. Tuttavia, raccomandiamo ai principiamti di scrivere il loro primo programma usando solo le istruzioni della Tabella 3-1. Questo vi aiuterà a superare meglio le difficoltà legate al dover imparare, allo stesso tempo, il set d'istruzioni dell'MC68000 e le basi della programmazione in linguaggio assembly. Una volta che avrete acquisito una certa esperienza, vi servirete anche delle altre istruzioni (Tabelle 3-2 e 3-3).

Tabella 3-1 - Istruzioni dell'MC68000 usate frequentemente.

Mnemonico d'istruzione	Significato	Mnemonico d'istruzione	Significato
ADD	Somma	JMP	Salto
AND	And logico	JSR	Salto una Subroutine
ASL	Shift aritmetico a sinistra	LSL	Shift logico a sinistra
ASR	Shift aritmetico a destra	LSR	Shift logico a destra
B_{CC}	Branch condizionato	MOVE	Trasferimento
BRA	Branch incondizionato	OR	OR logico
BSR	Branch a una subroutine	ROL	Rotazione a sinistra
CLR	Pone l'operando a 0	ROR	Rotazione a destra
CMP	Confronto	RTS	Ritorno da una subroutine
EOR	OR esclusivo	SUB	Sottrazione

Sono elencate unicamente le versioni delle istruzioni che utilizzano operandi della lunghezza di una word. Le versioni delle stesse istruzioni che utilizzano operandi di un byte o di una long word, se disponibili, sono utilizzate con la stessa frequenza.

Tabella 3-2 - Istruzioni dell'MC68000 usate solo occasionalmente.

Mnemonico d'istruzione	Significato	Mnemonico d'istruzione	Significato
ABCD	Somma decimale con riporto	NEG	Negazione
BTST	Testa lo stato di un bit	NOP	Nessuna operazione
DB_{CC}	Testa la condizione, decrementa ed esegue una diramazione (branch)	ROXL	Rotazione a sinistra con riporto
		ROXR	Rotazione a destra con riporto
EXG	Scambio del contenuto di due registri		
		RTE	Ritorno da una exception
MOVEM	Trasferimento tra registri e memoria	RTR	Ritorno e Restore
MOVEP	Trasferimento tra processore e periferiche	SBCD	Sottrazione decimale con riporto
MULS	Moltiplicazione tra numeri con segno	STOP	Stop
		SWAP	Scambio di bit in un registro
MULU	Moltiplicazione tra numeri senza segno	TST	Test

Sono elencate unicamente le versioni delle istruzioni che utilizzano operandi della lunghezza di una word. Le versioni delle stesse istruzioni che utilizzano operandi di un byte o di una long word, se disponibili, hanno la stessa frequenza di utilizzo

Tabella 3-3 - Istruzioni dell'MC68000 usate molto raramente.

Significato	Mnemonico d'istruzione		
	Mnemonico d'istruzione	Significato	
BCHG	Test su un bit e cambio del suo valore	LINK	Crea un link con lo stack
		NBCD	Negazione di un decimale
BCLR	Testa un bit e lo pone a 0	PEA	Caricamento di un indirizzo nello stack
BSET	Testa un bit e lo pone a 1	RESET	Reset di un device
CHK	Controllo del valore di un registro		
		S_{CC}	Riempie di 1 un registro se la condizione è vera
DIVS	Divisione con segno		
DIVU	Divisione senza segno	TAS	Test and Set
EXT	Estensione del bit di segno	TRAP	Inizia il trattamento di una exception
LEA	Caricamento di un indirizzo	TRAPV	Trattamento di un overflow
		UNLNK	Toglie il link con lo stack

Sono elencate unicamente le versioni delle istruzioni che utilizzano operandi della lunghezza di una word. Le versioni delle stesse istruzioni che utilizzano operandi di un byte o di una long word, se disponibili, sono utilizzate con la stessa frequenza.

MODI OPERATIVI DELL'MC68000

L'MC68000 può operare sia in modo Supervisore (o di Sistema) che in modo Utente (o normale). Il valore di un flag di stato consente di selezionare un modo o l'altro.

Alcune istruzioni possono essere eseguite solo nel modo Supervisore. Sono presenti anche due diversi puntatori di stack, che permettono di avere due stack differenti, uno per il modo Supervisore, l'altro per il modo Utente.

A coloro che ancora non hanno molta familiarità con il linguaggio assembly, consigliamo di mantenere l'MC68000 nel modo supervisore, ignorando il modo Utente, così da poter disporre di tutte le istruzioni. Eviterete di imbattervi in istruzioni eseguite solo in modo Supervisore, il che finirebbe per confondervi ulteriormente.

Ma c'è una buona ragione che giustifica la presenza di due diversi modi operativi. Come vi potrà dire ogni programmatore esperto, i programmi in linguaggio assembly si suddividono in *software di sistema e programmi applicativi.* Al primo gruppo appartengono quei programmi che servono a coordinare le varie componenti di un sistema computerizzato e che, se necessario, possono essere scritti in modo Supervisore. I programmi applicativi svolgono una determinata funzione stabilita dall'utente e dovrebbero essere scritti sempre in modo Utente.

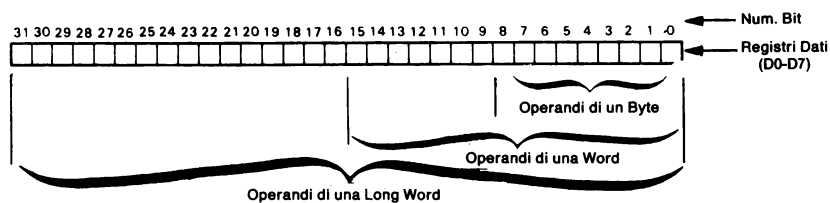
REGISTRI E FLAG DELL'MC68000

Registri dati

L'MC68000 dispone di otto registri dati, di sette registri indirizzi, di due puntatori di stack e di un contatore di programma, tutti a 32 bit, oltre ad un registro di stato a 16 bit. La Figura 3-1 mostra tutti i registri dell'MC68000.

Il registro di stato dell'MC68000 contiene cinque flag di stato, tre bit per la maschera di interrupt, un bit per la selezione del modo operativo (Supervisore o Utente) ed un bit per attivare il modo Trace. I cinque flag di stato sono:

Carry (C)
Overflow (V)
Zero (Z)
Negative (N)
Extend (E)

Questi flag occupano i cinque bit meno significativi del registro di stato, com'è indicato nella Figura 3-2.

I REGISTRI DELL'MC68000

Gli otto registri dati possono essere usati per gestire operandi di 8 bit (byte), di 16 bit (word) o di 32 bit (long word). L'illustrazione seguente mostra la posizione occupata da operandi di dimensioni diverse all'interno di un registro dati.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
D0
D1
D2
D3
D4
D5
D6
D7
Registri Dati
31 16 15 0
A0
A1
A2
A3
A4
A5
A6
Registri Indirizzi
Puntatore Stack Utente
Puntatore Stack Supervisore
A7
Puntatori di stack
31 24 23
Contatore di Programma
15 8 7 0
Byte di Sistema
Byte Utente
Registro di Stato

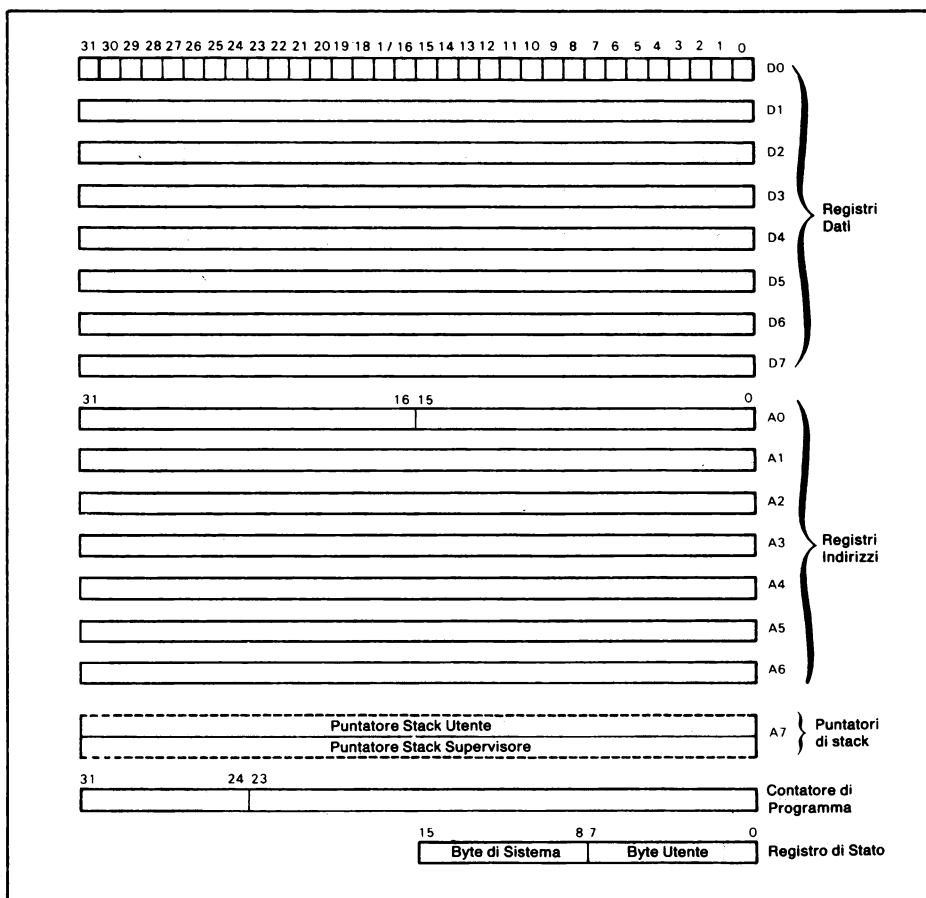



Figura 3-1. Registri Programmabili dell'MC68000

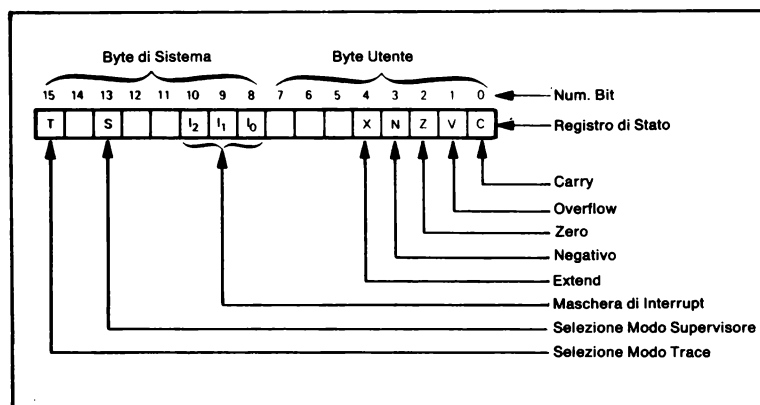



Figura 3-2. Il Registro di Stato dell'MC68000

Tutti i registri dati sono dei normali accumulatori che possono, inoltre, essere impiegati come registri indice o contatotri. Si dispone di una completa flessibiltà nell'utilizzazione di questi registri, poichè nessuno di essi è riservato esclusivamente ad una particolare funzione.

Registri indirizzi

Ci sono sette registri indirizzi generici (A0-A6), che sono in grado di gestire operandi di lunghezza pari ad una word (16 bit) o ad una long word (32 bit), ma non possono essere impiegati con operandi da 8 bit. Come indica il loro nome, i registri indirizzi sono normalmente utilizzati per contenere degli indirizzi, anzichè dei dati, e possono anche essere impiegati come indice nell'indirizzamento indicizzato della memoria. In quest'ultimo caso, la loro funzione è quella tipica dei registri indice di un microcomputer, che trovate descritta in *An Introduction to Microcomputers: Volume 1*[2].

Puntatore allo stack

Il registro indirizzi A7, oltre a servire come normale registro indirizzi o registro indice, svolge la funzione di puntatore di stack. Il registro A7 è costituito, in realtà, da due registri diversi, uno utilizzato in modo Supervisore, l'altro in modo Utente. Perciò, se mettiamo un dato in A7 quando ci troviamo in modo Supervisore, passando in modo Utente non sarà possibile rileggerlo. Questo consente di avere a disposizione uno stack riservato per ciascuno dei due modi operativi.

Contatore di Programma

Il contatore di programma è un tipico contatore di programma, come è descritto nel primo volume di *An Introduction to Microcomputers*.

Sebbene i registri indirizzi ed il contatore di programma contengano 32 bit, per indirizzare la memoria ne vengono impiegati solo 24. Gli otto bit di ordine alto sono completamente ignorati in fase di indirizzamento.

IL REGISTRO DI STATO

L'MC68000 dispone di un registro di stato a 16 bit, suddiviso in un byte di Sistema ed in un byte Utente. La funzione dei bit del registro di stato è indicata nella Figura 3-2.

Il bit di Carry contiene il riporto del bit più significativo, in seguito ad operazioni aritmetiche o di shift. Al pari di molti microprocessori, l'MC68000 inverte il valore reale del riporto dopo una sottrazione, per cui **il bit di Carry agisce come bit di prestito. Nell'MC68000, le operazioni logiche azzerano il flag di Carry e altrettanto fanno le istruzioni di trasferimento, moltiplicazione e divisione.**

Il bit di Zero è standard. È posto a 1 quando un'operazione dà come risultato zero; viene azzerato nel caso di un risultato diverso da zero.

Il bit di Negativo (segno) è, anch'esso, standard ed assume il valore del bit più significativo di un risultato. Un valore 1 nel bit di Negativo indica un risultato negativo, mentre un valore 0 indica un risultato positivo: questo nel caso di numeri in complemento a due provvisti di segno. Qualora siano impiegati numeri assoluti, questo bit può essere completamente ignorato oppure usato per conoscere il valore del bit più significativo del risultato.

Anche il bit di Overflow è del tipo standard, descritto nel volume 1 di *An Introduction to Microcomputers*. Viene posto a 1 tutte le volte che il risultato di un'operazione è maggiore di quello rappresentabile in un registro. Il processore mette a 1 il flag di Overflow quando il riporto del bit più significativo è diverso da quello del bit subito accanto; in pratica, l'overflow è l'OR esclusivo dei riporti dei due bit di ordine più alto. Nell'MC68000 le operazioni logiche azzerano il flag di Overflow, come fanno le istruzioni di spostamento, di rotazione e molte altre.

Il bit di Extend assume sempre lo stesso valore del bit di Carry, ogni volta che questo viene modificato da un'istruzione. È utilizzato nelle operazioni aritmetiche in precisione multipla.

Molte istruzioni modificano i bit di stato, anche quando tali modifiche non sono importanti ai fini dell'operazione eseguita. Consigliamo, quindi, di consultare la tabella con il riepilogo delle istruzioni nell'Appendice A, per stabilre come un particolare bit di stato venga modificato da una determinata istruzione.

Il byte di Sistema del registro di stato contiene informazioni relative al sistema, al contrario del byte Utente, correlato alle istruzioni. I bit del byte di sistema possono essere modificati solo quando l'MC6800 si trova in modo Supervisore.

I tre bit meno significativi del byte di sistema del registro di stato forniscono la maschera di priorità degli interrupt. L'MC68000 dispone di tre linee di interrupt, consentendo di codificare fino a sette livelli di priorità. La maschera di interrupt stabilisce quali livelli saranno riconosciuti dal processore. Se, ad esempio, poniamo la maschera di interrupt a 100, i livelli dallo 0 al 4 sono disattivati e le richieste di interrupt con quelle determinate sequenze di bit (000, 001, 010, 011, 100) saranno ignorate.

Il bit S del registro di stato è utilizzato per passare dal modo Supervisore a quello Utente e viceversa. Quando il suo valore è 1 il processore opera in modo Supervisore, quando è 0 in modo Utente. Va ricordato che ciascuno di questi modi operativi ha il proprio puntatore di stack e che alcune istruzioni privilegiate possono essere eseguite solo in modo Supervisore.

Il bit T del registro di stato serve a porre l'MC68000 in modo trace. Di questo parleremo più diffusamente nel Capitolo 19.

LA MEMORIA DELL'MC68000

La memoria dell'MC68000, al pari dei registri, è organizzata in byte, word e long word. L'indirizzo di ogni byte è rappresentato da un numero di 24 bit e può assumere un qualsiasi valore. Gli indirizzi delle word e delle doppie word devono essere numeri pari. Nelle illustrazioni di questo libro mostremo la memoria suddivisa in parole, ciascuna formata da due byte. L'indirizzo compare a destra di ogni word e corrisponde a quello del byte di ordine alto. L'indirizzo del byte di ordine basso è maggiore di 1.

MODI DI INDIRIZZAMENTO

Le istruzioni del linguaggio assembly dicono al processore quale operazione eseguire e quali indirizzi utilizzare, cioè dove trovare i dati su cui operare. La parte dell'istruzione che dice al processore quale operazione svolgere è il *codice operativo*. Nell'Appendice C sono elencati i codici operativi mnemonici ed i loro equivalenti numerici. La parte dell'istruzione contenente le informazioni relative agli indirizzi da usare è *il campo dell'operando o degli indirizzi*. Il processore se ne serve per sapere dove trovare gli operandi e dove mettere i risultati.

CARATTERISTICHE GENERALI

Tipi di indirizzamento diretto a registro

Esistono varie forme per indicare al processore quali indirizzi deve usare: sono i cosiddetti *modi di indirizzamento*. Prima di analizzarli uno ad uno, soffermiamoci brevemente sulle loro caratteristiche generali. I due modi seguenti non coinvolgono assolutamente la memoria:

1. **Indirizzamento intrinseco:** è sufficiente il solo codice operativo ad indicare al processore ciò che deve fare.
2. **Indirizzamento a registro:** l'operando è contenuto in uno dei registri.

Questi sono, invece, i più comuni modi di indirizzamento che riguardano anche la memoria :

3. **Indirizzamento immediato:** l'operando si trova immediatamente dopo il codice operativo nella memoria di programma.
4. **Indirizzamento diretto:** l'indirizzo da usare segue il codice operativo nella memoria di programma.
5. **Indirizzamento indicizzato:** l'indirizzo effettivo è la somma di un indirizzo di base e di un indice (offset).
6. **Indirizzamento indiretto:** l'indirizzo da usare si trova in un registro o in memoria; cioè, l'istruzione indica al processore dove trovare l'indirizzo del dato e non il dato vero e proprio.
7 **Indirizzamento relativo:** l'operando è posto ad una certa distanza rispetto al valore attuale del contatore di programma.

Il Capitolo 6 di *An Introduction to Microcomputers: Volume 1* descrive tutti questi modi di indirizzamento e le relative combinazioni.

Modi di Indirizzamento dell' MC6800

L'MC68000 ha un insieme potente e versatile di modi di indirizzamento. I modi disponibili sono elencati qui di seguito, nell'ordine in cui li prenderemo in esame:

1. **Operando intrinseco** (istruzioni che non richiedono indirizzi)
2. **Registri come operandi** (istruzioni che usano solo i contenuti dei registri come operandi)

Gli altri tipi specificano degli indirizzi di memoria; essi sono:

3. **Immediato**
4. **Assoluto o diretto**
5. **Indiretto a registro indirizzi**
6. **Indiretto a registro indirizzi con spostamento**
7. **Indiretto a registro indirizzi con postincremento**
8. **Indiretto a registro indirizzi con predecremento**
9. **Indiretto a registro indirizzi con indice e spostamento**
10. **Relativo al contatore di programma con spostamento**
11. **Relativo al contatore di programma con indice e spostamento.**

INDIRIZZO EFFETTIVO

Nel descrivere i vari tipi di indirizzamento ed il modo in cui utilizzarli, faremo riferimento molto spesso all'indirizzo reale, quello di cui il processore, in definitiva, si serve per eseguire una determinata operazione. È il cosiddetto *indirizzo effettivo,* dove il processore trova

l'operando o dove mette un risultato. In alcuni casi (come, ad esempio, nell'indirizzamento immediato) l'indirizzo effettivo è semplicemente la locazione che segue il codice operativo dell'istruzione. In altri, la determinazione dell'indirizzo effettivo non risulta così semplice, in quanto esso fa addirittura parte di un'istruzione oppure è contenuto in un registro o in una locazione di memoria. Per stabilire l'indirizzo effettivo sono necessari, molte volte, dei calcoli piuttosto complessi, come sommare un certo valore ad un registro, ecc. Alcuni indirizzamenti sono difficili da capire, dal momento che richiedono una serie di operazioni che, solo alla fine, forniscono l'indirizzo effettivo. Spiegheremo, successivamente, la loro utilità e ne descriveremo alcune applicazioni pratiche. Quello che è importante è che abbiate ben chiari i vari metodi di indirizzamento, perchè questa è la chiave per scrivere programmi potenti e, allo stesso tempo, sufficientemente generici. Ricordate che il processore riesce sempre a stabilire con esattezza l'indirizzo effettivo, anche quando sono necessarie operazioni molto complesse.

MODI DI INDIRIZZAMENTO CHE NON SPECIFICANO LOCAZIONI DI MEMORIA

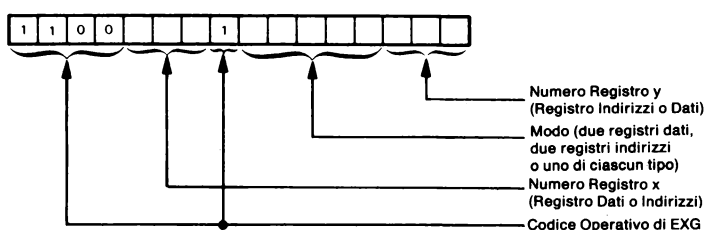
INDIRIZZAMENTO INTRINSECO

In questo caso è lo stesso codice operativo che indica al processore quale indirizzo utilizzare. Ad esempio, RTE (Return from Exception), RTS (Return from Subroutine), RTR (Return and Restore Condition Codes) costringono il processore a servirsi del puntatore di stack per trasferire dati da o verso la memoria. Analogamente, l'istruzione TRAPV (Trap on Overflow) fa sì che il processore usi il puntatore dello stack Supervisore per salvare in memoria il contenuto del contatore di programma e ottenga da un indirizzo prestabilito il vettore per l'operazione di Trap. NOP (No Operation) e RESET non hanno bisogno di nessun operando. I codici operativi da soli sono sufficienti, non è necessaria nessun'altra informazione.

I testi della Motorola raggruppano le istruzioni precedenti in una categoria di indirizzamento che definiscono implicito. A questa categoria dovrebbero appartenere tutte quelle istruzioni che fanno riferimento implicitamente ad un qualsiasi registro dell'MC68000: le istruzioni Branch, che agiscono sul contatore di programma, le istruzioni Move, che alterano il contenuto del registro di stato o del puntatore di stack, e le istruzioni Jump, che cambiano il valore presente nel contatore di programma. Tuttavia, non si può parlare, in questo caso, di indirizzamento intrinseco nel vero senso della parola, dal momento che queste istruzioni necessitano di ulteriori informazioni, non essendo sufficienti quelle fornite dal codice operativo.

INDIRIZZAMENTO A REGISTRO

Con molte istruzioni dell'MC68000 è possibile usare operandi contenuti nei registri del processore e non è necessario, perciò, un indirizzamento della memoria. Per altre è *obbligatorio* servirsi dei registri come operandi. Infatti, EXG (Exchange Registers), EXT (Sign Extend), SWAP (Swap Register Values) ed alcune istruzioni MOVE agiscono solo su operandi contenuti in un registro o non possono mai utilizzare operandi contenuti in memoria.

L'istruzione EXG scambia il contenuto di due registri dati o indirizzi. È un'operazione che agisce su operandi di grandezza pari ad una long word (32 bit) e perciò è l'intero contenuto dei due registri ad essere scambiato. Due campi da 3 bit, all'interno del codice operativo dell'istruzione, definiscono i numeri dei registri e un campo di "modo" specifica se si tratta di registri dati, di registri indirizzi o di un registro dati e di uno indirizzi. Il codice operativo dell'istruzione EXG può essere rappresentato nel modo seguente:

Per maggiori dettagli sulle modalità di codifica del registro vi rimandiamo alla descrizione dell'istruzione EXG, nel Capitolo 22.

Mentre solo poche istruzioni *devono* usare i contenuti dei registri come operandi, la maggior parte *permette* di indicare operandi contenuti nei registri. **Esistono due modi di indirizzamento diretto a registro. L'unica differenza tra i due è naturalmente che il primo usa un registro dati come operando, mentre il secondo si serve di un registro indirizzi.**

Alcune istruzioni che consentono l'impiego dell'indirizzamento diretto a registro, richiedono che almeno uno dei registri sia un registro dati, mentre per altre è necessario avere almeno un registro indirizzi. Ad esempio, l'istruzione ADD (Add Binary) richiede che uno degli operandi sia contenuto in un registro dati, mentre l'altro può essere sia in un registro dati (indirizzamento diretto a registro dati) che in un registro indirizzi (indirizzamento diretto a registro indirizzi). **La Figura 3-3 illustra l'indirizzamento diretto a registro dati con una istruzione Add Binary (ADD).** Entrambi gli operandi utilizzati in questa operazione di ADD si trovano in due registri dati; non c'è necessità di far riferimento ad operandi contenuti in memoria.

Figura 3-3. Indirizzamento Diretto a Registro Dati

La Figura 3-4 mostra la stessa istruzione ADD, questa volta con indirizzamento diretto a registro indirizzi. Come abbiamo detto sopra, questa istruzione richiede che uno degli operandi sia contenuto

Figura 3-4. Indirizzamento Diretto a Registro Indirizzi

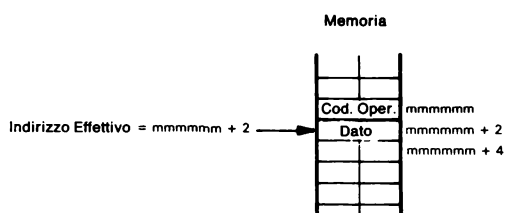
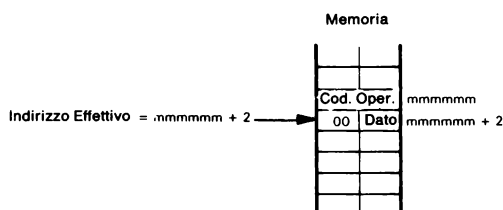
in un registro dati (in questo caso, D3). Dato che, in questa occasione, ci siamo serviti dell'indirizzamento diretto a registro indirizzi, l'altro operando viene prelevato da un registro indirizzi (A6).

L'MC68000 ha un'altra istruzione simile (ADDA - Add Address) che deve trovare almeno uno degli operandi in un registro indirizzi, il quale conterrà anche il risultato della somma. Il secondo operando può essere un registro indirizzi o un registro dati. (Entrambe queste istruzioni di somma prevedono, inoltre, la possibilità che il secondo operando sia in una locazione della memoria; tuttavia, dal momento che adesso stiamo parlando dell'indirizzamento diretto a registro, preferiamo considerare questo aspetto in seguito, quando descriveremo gli altri modi di indirizzamento).

MODI DI INDIRIZZAMENTO DELLA MEMORIA

INDIRIZZAMENTO IMMEDIATO

Sintassi dell'indirizzamento immediato

Nell'indirizzamento immediato i dati seguono immediatamente il codice operativo. Questo significa che l'indirizzo effettivo è semplicemente il contenuto del contatore di programma una volta che il processore ha prelevato il codice operativo. Lo possiamo rappresentare in questo modo:

L'MC68000 dispone di istruzioni immediate della lunghezza di un byte, di una word o di una long word. Nei primi due casi, l'operando si trova nella word successiva al codice oggetto dell'istruzione, com'è indicato nella figura precedente. Naturalmente, se il dato è di un byte, esso ne occuperà la metà di ordine basso, mentre quella di ordine alto deve contenere solo degli zeri, come appare dallo schema seguente:

Nel linguaggio assembly dell'MC68000 l'indirizzamento immediato viene indicato facendo precedere l'operando dal simbolo #. Ad esempio, l'assemblatore dell'MC68000 converte lo statement

```
ADD     #$1066, D3
        (# significa "indirizzamento immediato" e
        $ significa "esadecimale")
```

in una istruzione ADD, che somma il valore 1066_{16} al registro dati D3. La Figura 3-5 ne descrive l'esecuzione.

Il registro D3 contiene inizialmente il valore 1848_{16}. Dopo che il processore ha eseguito l'istruzione ADD #$1066,D3 il contenuto del registro D3 sarà $1848_{16} + 1066_{16} = 28AE_{16}$. Il processore incrementa quattro volte il contatore di programma, due volte subito dopo aver prelevato il codice operativo e due volte quando ha prelevato il dato immediato, in questo caso 1066_{16}.

Dimensione dei dati

Nell'esempio che vi abbiamo mostrato, l'istruzione ADD agisce su una parola di 16 bit, che è la lunghezza di default per un dato. La maggior parte delle istruzioni, comunque, è in grado di operare su dati di dimensioni differenti. Volendo specificare una lunghezza diversa da quella di default (che è appunto di una word) bisogna far seguire al codice operativo di un'istruzione un codice che rappresenti la lunghezza del dato. L'assemblatore standard dell'MC68000 consente di indicare la dimensione di un dato mediante l'aggiunta di una lettera al mnemonico di un'istruzione: B indica un byte (un dato di 8 bit), W indica una word (un dato di 16 bit), L una long word (un dato da 32 bit). Questi codici seguono il codice operativo, dal quale devono essere separati tramite un punto (.). Ad esempio, l'istruzione ADD

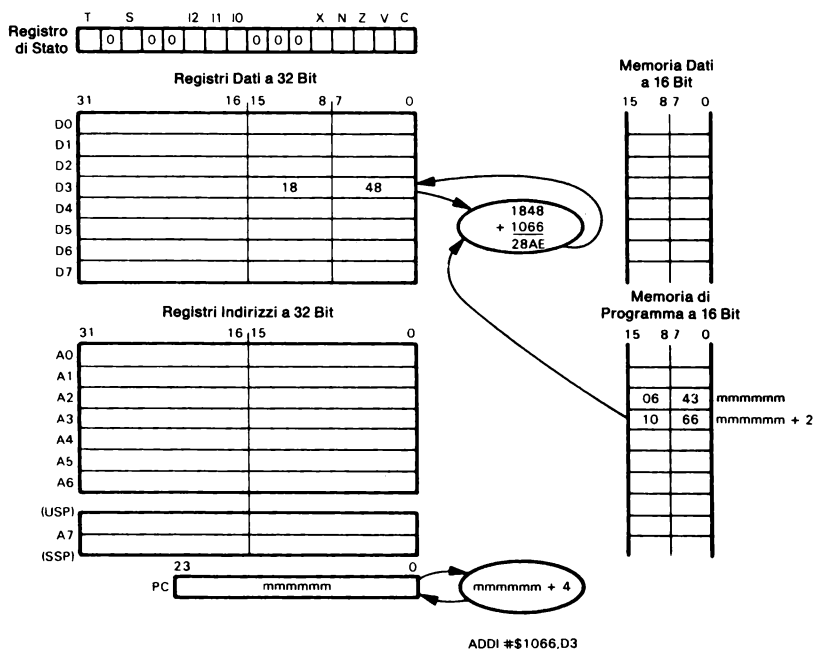



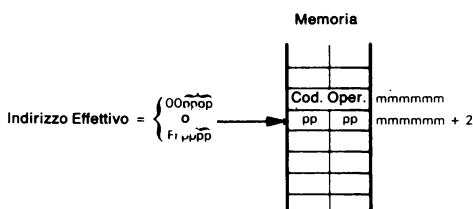
Figura 3-5. Indirizzamento Immediato

della figura precedente poteva essere scritta nella forma ADD.W #$1066,D3. Non è necessario, comunque, specificare la lunghezza con un dato da 16 bit (word), poichè questa è l'opzione di default.

Indirizzamento immediato rapido

L'MC68000 dispone anche di uno modo speciale ("rapido") di indirizzamento immediato per operandi piccoli in cui il dato è, in realtà, contenuto nella word del codice operativo. ADDQ (Add Quick) e SUBQ (Subtract Quick) servono, rispettivamente, per sommare o sottrarre dei numeri compresi tra uno e otto. MOVEQ (Move Quick) può essere usata per trasferire in un registro o in una locazione di memoria dei valori compresi tra –128 e +127.

INDIRIZZAMENTO (DIRETTO) ASSOLTO CORTO

In questo caso la metà di ordine di basso dell'indirizzo effettivo segue, in memoria, il codice operativo. La metà di ordine alto si ottiene mediante estensione del bit di segno (bit 15) della metà di ordine basso dell'indirizzo stesso. In questo modo, è possibile ottenere indirizzi a 32 bit compresi fra 000000_{16} e $007FFF_{16}$ e fra $FFFF8000_{16}$. L'indirizzamento assoluto (diretto) corto è descritto dallo schema seguente:

Gli indirizzi assoluti corti compresi fra 0000_{16} e $7FFF_{16}$ si riferiscono alle corrispondenti locazoni di memoria, mentre **quelli compresi fra 8000_{16} e $FFFF_{16}$ si riferiscono ai più alti indirizzi disponibili, proprio grazie alla estensione del segno.**

È opportuno notare che mentre i testi sull'MC68000 lo definiscono *indirizzamento assoluto corto*, il volume 1 di *An Introduction to Microcomputers* lo indica come *modo diretto a pagina base*. In questo caso la pagina base è costituita dai 64K di memoria più bassi e dai 64K più alti. Questo è un modo rapido e breve per usare routine e dati che occupano locazioni della pagina base, risparmiando una word nella memoria di programma e riducendo di un ciclo di lettura il tempo di esecuzione.

L'assemblatore standard dell'MC68000 usa l'indirizzamento assoluto corto tutte le volte che è disponibile e non è stato specificato un altro modo. Naturalmente purché l'indirizzo si trovi entro i limiti previsti.

La Figura 3-6 mostra l'uso dell'indirizzamento assoluto corto con un'istruzione ADD, che somma il contenuto della locazione di me-

moria 007100_{16} a quello del registro dati D3. Una volta che il processore ha eseguito l'istruzione il valore presente nel registro D3 sarà $1234_{16} + (007100_{16}) = 1234_{16} + 5678_{16}$. Il processore incrementa il contatore di programma quattro volte; due dopo aver prelevato il codice operativo e due dopo aver prelevato l'indirizzo diretto.

L'indirizzo assoluto corto occupa solo una word, anche se l'istruzione (come ad es. ADD.L) agisce su operandi da 32 bit. In tal caso, il processore utilizza gli indirizzi 007100_{16} e 007102_{16} (del nostro esempio) per prelevare le word del dato, rispettivamente, di ordine alto e di ordine basso.

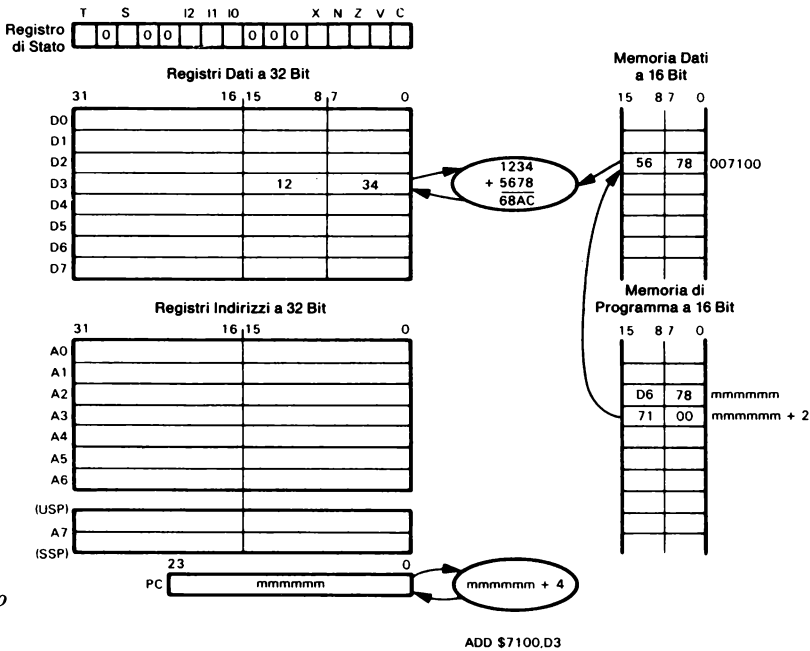
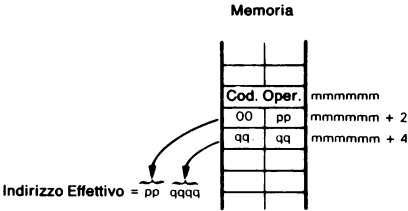



Figura 3-6. Indirizzamento (Diretto) Assoluto Corto

INDIRIZZAMENTO (DIRETTO) ASSOLUTO LUNGO

L'indirizzo effettivo occupa le due word della memoria di programma che seguono il codice operativo, con la metà di ordine alto posta nella prima word: questo è il formato standard dell'MC68000. L'indirizzamento diretto assoluto lungo può essere schematizzato in questo modo:

Si deve notare che i testi che trattano dell'MC68000 lo definiscono *indirizzamento assoluto lungo*, mentre invece il volume 1 di *An Introduction to Microcomputers* lo indica come *modo diretto o diretto esteso*.

Questo tipo di indirizzamento consente al processore di accedere ad una locazione di memoria qualsiasi. Naturalmente, non è necessario utilizzarlo per locazioni che si trovano nei 64K più alti o più bassi della memoria e per le quali è disponibile quello assoluto corto. Il modo assoluto lungo è quello impiegato abitualmente per indirizzi che non sono nel raggio d'azione dell'indirizzamento assoluto corto e, nell'assemblatore standard dell'MC68000, rappresenta l'opzione di default.

La Figura 3-7 mostra un'istruzione ADD con indirizzamento assoluto lungo. È una situaione analoga a quella descritta per l'indirizzamento assoluto corto, tranne per il fatto che dopo il codice operativo si trova un indirizzo di 32 bit.

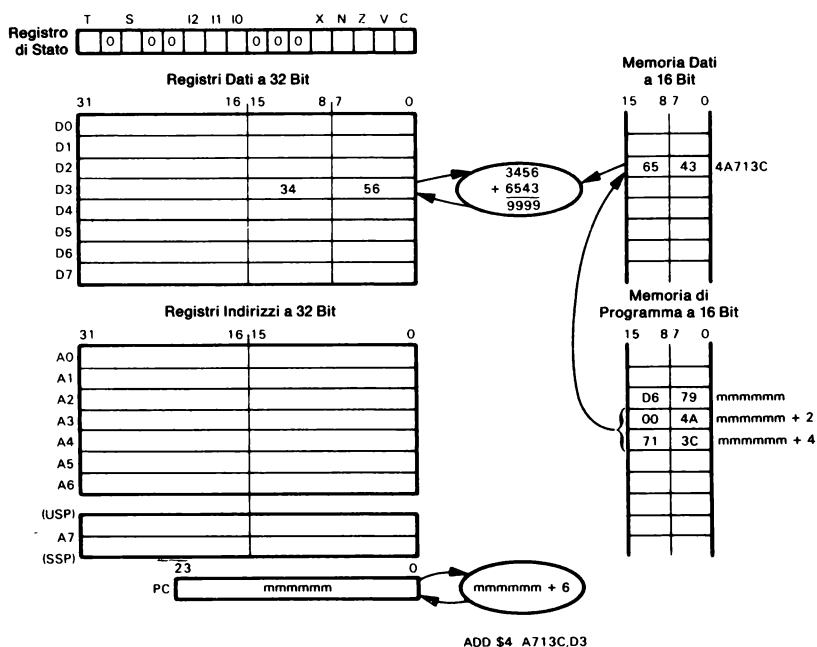



Figura 3-7. Indirizzamento (Diretto) Assoluto Lungo

INDIRIZZAMENTO INDIRETTO A REGISTRO INDIRIZZI

Sintassi dell'indirizzamento indiretto

L'indirizzo dell'operando è contenuto in un registro indirizzi. Bisogna rilevare che questo non è il vero e proprio indirizzamento indiretto, nel quale è una locazione di memoria, e non un registro, a fornire l'indirizzo effettivo. In *An Introduction to Microcomputers: Volume 1*

questo tipo di indirizzamento è definito "implied". L'MC68000 non prevede un indirizzamento indiretto che utlizzi una locazione di memoria.

Nel formato standard dell'assemblatore dell'MC68000 l'indirizzamento indiretto a registro viene specificato mettendo fra parentesi l'indicazione del registro indirizzi: ad esempio (A3). L'assemblatore trasforma uno statement

ADD (A3),D3

in un'istruzione ADD che somma il contenuto della locazione di memoria indicata in A3 al contenuto di D3. La Figura 3-8 mostra l'indirizzamento indiretto a registro indirizzi con un'istruzione ADD, che si serve dell'indirizzo ppqqqq, contenuto nel registro A3, per ottenere l'operando posto nella memoria dati.

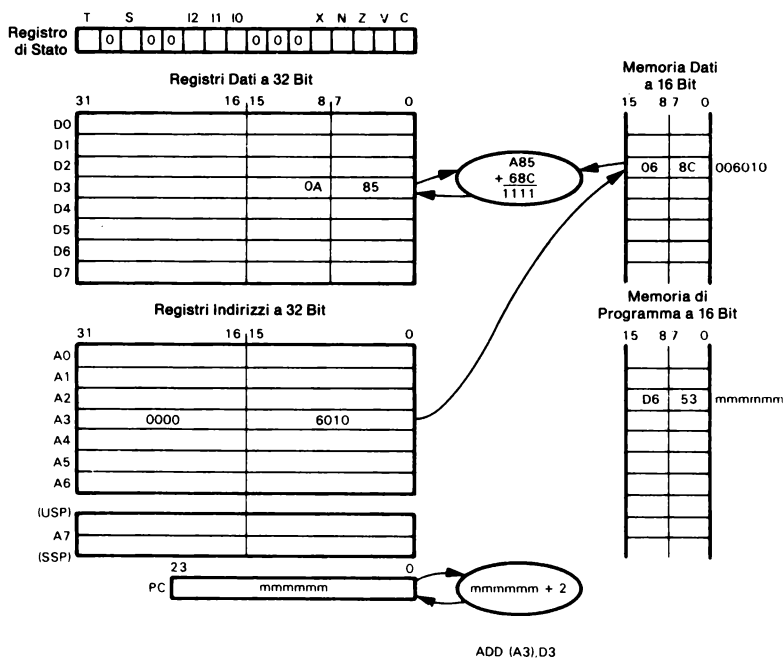



Figura 3-8. Indirizzamento Indiretto a Registro Indirizzi

AUTOINCREMENTO E AUTODECREMENTO

Nella elaborazione di array, di stringhe o di liste spesso è necessario elaborare un byte o una word per volta e passare, subito dopo, al byte o alla word seguenti che occupano il successivo indirizzo di memoria più alto (se ci stiamo muovendo in avanti) o l'indirizzo più basso (se ci spostiamo indietro). Volendo stampare una stringa di caratteri biso-

gnerà inviare alla stampante un carattere per volta. Analogamente, se vogliamo calcolare la media di dieci valori letti dobbiamo sommarli insieme uno ad uno (iniziando, per esempio, con 0 ed aggiungendo il primo valore, poi il secondo e così via) dividendo per dieci il risultato ottenuto.

Così, per elaborare un byte e spostarsi in avanti, dovremo:

- Ottenere il byte, usando il contenuto di un registro indirizzi.
- Aggiungere uno al registro indirizzi per fargli indicare il byte successivo.

L'effetto è simile a quello di una macchina da scrivere che stampa il carattere corrispondente al tasto premuto e, allo stesso tempo, sposta il carrello alla posizione successiva. Sottrarre uno dal registro indirizzi sarebbe come spostare indietro il carrello di una macchina da scrivere, a differenza della quale un computer non ha nessuna preferenza riguardo alla direzione dello spostamento.

Variazioni dell' Autoincremento e Autodecremento

Possibili valori degli incrementi e dei decrementi

L'MC68000 consente incrementi e decrementi di varia grandezza. Il registro indirizzi di base può essere:

- Incrementato di uno, dopo che è stato usato
- Incrementato di due, dopo che è stato usato
- Incrementato di quattro, dopo che è stato usato
- Diminuito di uno, prima di essere usato
- Diminuito di due, prima di essere usato
- Diminuito di quattro, prima di essere usato

Un incremento o un decremento di due è usato nel caso di un array formato da indirizzi o dati a 16 bit, mentre incrementi o decrementi di quattro sono riservati ad array di dati o indirizzi a 32 bit. Il processore si sposta automaticamente all'elemento successivo, anche se si trova a due o a quattro byte di distanza dall'elemento attuale. Applicando l'incremento dopo aver usato la base ed il decremento prima di usarla si garantisce la compatibilità con l'uso automatico dei puntatori di stack (nelle istruzioni BSR, JSR, RTE, RTR, RTS e TRAP ed in caso di Exception). Sarebbe possibile qualsiasi tipo di sequenza accesso/cambiamento di puntatore, ma questo è il metodo più diffuso. Tutto quello che l'utente deve ricordare è di caricare il registro indirizzi di base con l'indirizzo iniziale dell'array o della stringa, in caso di autoincremento, e con l'indirizzo finale aumentato di 1, 2 o 4, in caso di autodecremento (poichè il primo autodecremento riduce il registro indirizzi di base, prima ancora di utilizzarlo).

Inizializzazione del registro di base

L'autoincremento e l'autodecremento rappresentano il metodo più semplice per elaborare degli array o delle stringhe, dal momento che

permettono di aggiornare automaticamente l'indirizzo di memoria utilizzato, durante l'esecuzione della stessa istruzione. Vi rimandiamo ai Capitoli 5 e 6 per un ulteriore approfondimento dei meccanismi di autoincremento e autodecremento.

La documentazione fornita dai costruttori dell'MC68000 definisce l'indirizzamento con autoincremento come *indirizzamento indiretto a registro indirizzi con postincremento,* mentre quello con autodecremento lo definisce *indirizzamento indiretto a registro indirizzi con predecremento.* Ciò significa che l'autoincremento e l'autodecremento possono essere usati solo con l'indirizzamento indiretto a registro indirizzi.

Sintassi dell'indirizzamento indiretto con postincremento

Nel formato standard dell'assemblatore per l'MC68000 l'indirizzamento indiretto a registro indirizzi con postincremento viene indicato mettendo fra parentesi il nome del registro indirizzi e facendolo seguire dal segno più: ad esempio, (A3)+. L'assemblatore converte lo statement

ADD (A3)+,D3

in una istruzione ADD che somma il contenuto della locazione di memoria, il cui indirizzo si trova in A3, con il valore presente nel registro D3. Una volta eseguita l'operazione di ADD, il contenuto di A3 sarà aumentato di due. Si noti che quando nessuna indicazione sulla dimensione accompagna l'istruzione ADD si presuppone una lunghezza pari ad una word. Il contenuto del registro A3 è stato, dunque, aumentato di due, dopo che è stata eseguita l'operazione di ADD. Se fosse stata specificata una lunghezza di un byte, allora A3

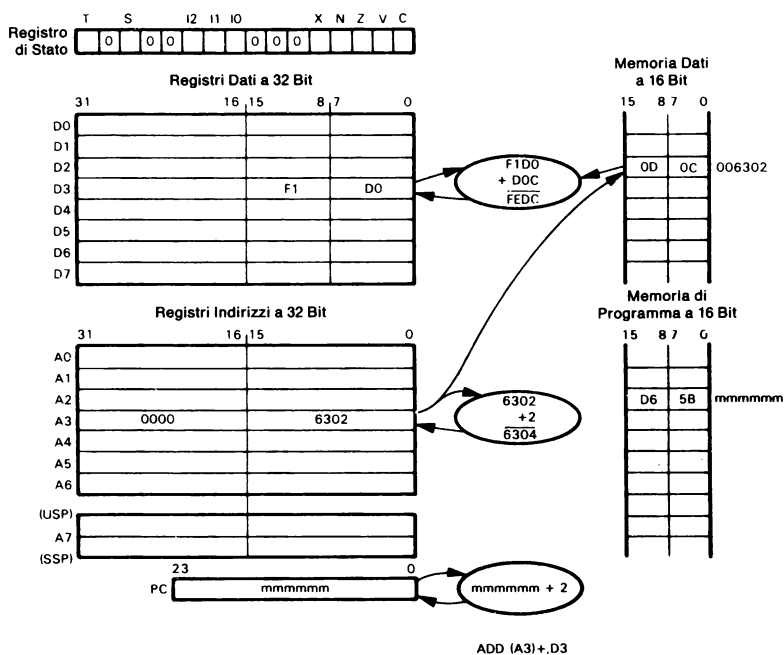



Figura 3-9 Indirizzamento Indiretto a Registro Indirizzi con Postincremento

sarebbe stato incrementato di uno, mentre, nel caso di una long word l'incremento di A3 sarebbe stato di quattro.

Sintassi dell'indirizzamento indiretto con predecremento

La Figura 3-9 mostra l'esecuzione dell'istruzione ADD, che fa uso dell'indirizzamento indiretto a registro indirizzi con postincremento.

Nel formato standard dell'assemblatore dell'MC68000, l'indirizzamento indiretto a registro indirizzi con predecremento viene indicato mettendo fra parentesi il registro indirizzi e facendolo precedere da un segno meno: ad esempio, –(A3).Quindi, l'assemblatore trasforma lo statement

```
ADD -(A3),D3
```

in una istruzione ADD che prima decrementa il contenuto del registro indirizzi A3 di due, quindi somma il contenuto della locazione di memoria, il cui indirizzo è contenuto in A3, al contenuto di D3.

La Figura 3-10 illustra l'esecuzione di questa istruzione ADD, che utilizza l'indirizzamento indiretto a registro indirizzi con predecremento.

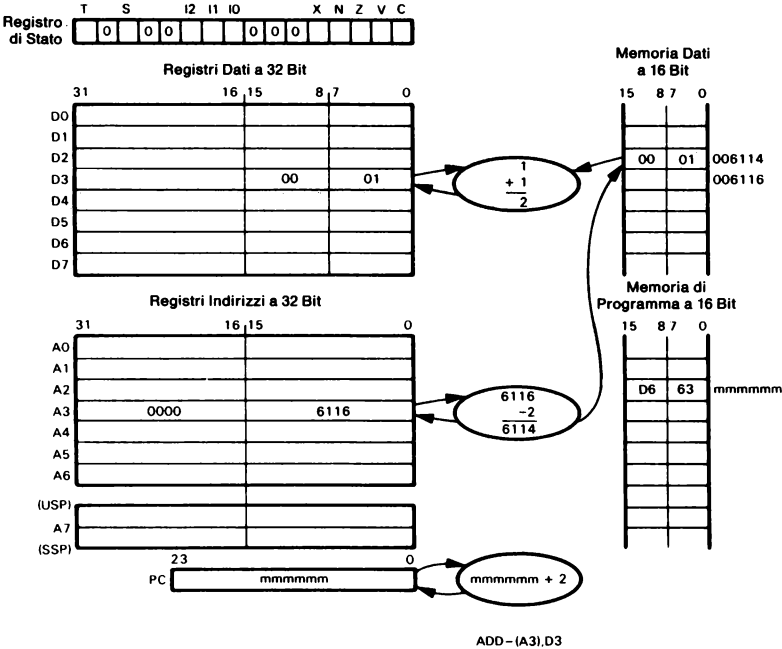



Figura 3-10. Indirizzamento Indiretto a Registro Indirizzi con Predecremento

INDIRETTO A REGISTRO INDIRIZZI CON SPOSTAMENTO (OFFSET)

Sintassi dell'indirizzamento indiretto con offset

In questo caso, l'indirizzo effettivo si ottiene sommando un determinato valore di spostamento (o offset) al valore contenuto in un registro indirizzi. L'offset segue il codice operativo ed è una costante, dal

momento che la memoria di programma normalmente non cambia durante l'esecuzione. Il contenuto di un registro indirizzi può variare poichè è il programma a modificarne il contenuto. Questo modo di indirizzamento permette di far riferimento ad un particolare elemento di un array o di una lista. Ad esempio, possiamo avere una serie di dieci temperature, corrispondenti ad altrettante misurazioni effettuate in un serbatoio; per cambiarne o mostrarne una in particolare, dobbiamo sapere dove questa serie ha inizio (indirizzo di base) e quale valore desideriamo (questo è l'offset). Se, come di solito avviene, queste temperature si trovano in locazioni di memoria consecutive, possiamo trovarne una servendoci di un offset costante rispetto all'indirizzo di base.

Allo stesso modo possiamo mettere in memoria un record formato dal nome di una persona, dal suo indirizzo, dal numero di codice fiscale, dall'età e dal tipo di lavoro. Se vogliamo avvisare di un cambiamento di stipendio tutte le persone che svolgono un determinato lavoro, possiamo ottenere il tipo di lavoro svolto specificando la sua distanza rispetto all'inizio del record (ad esempio, 97 byte più avanti). È un pò come dire a tutti gli studenti che stanno facendo

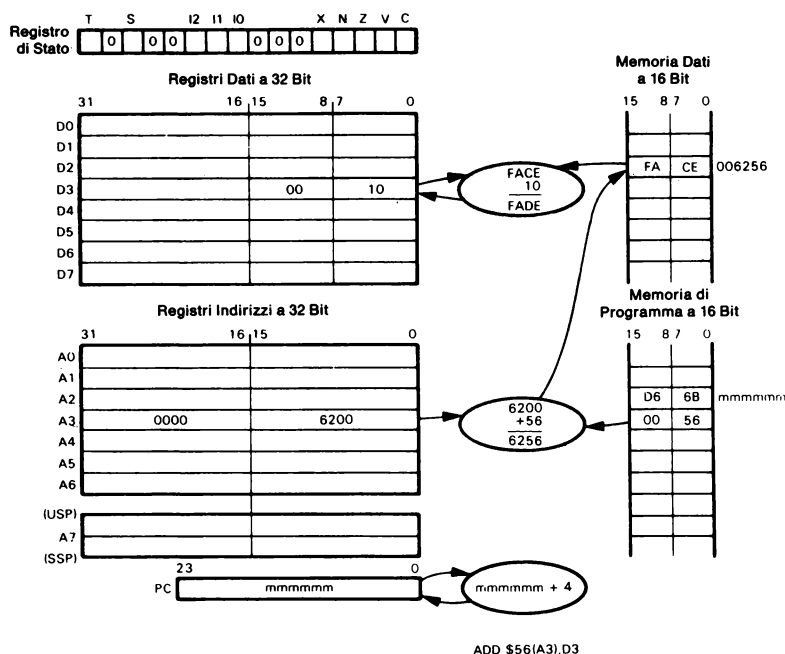



Figura 3-11. Indirizzamento Indiretto a Registro Indirizzi con Spostamento

un'esame di mettere i loro nomi sulla prima riga del foglio, la classe sulla quinta e la data di nascita sulla decima riga. Ogni studente è in grado di riconoscere la prima riga del suo foglio. Così, in un record, il nome può occupare i primi 24 byte, l'indirizzo i 70 byte successivi, il numero di codice fiscale gli altri 16 e l'età gli ultimi 3. Possiamo individuare un particolare campo di un record (per esempio il numero di codice fiscale dell'impiegata Sue), specificando l'indiriz-

zo di base (in questo caso, l'indirizzo dove inizia il record dell'impiegata Sue) e di quanto ci dobbiamo spostare per raggiungere il campo desiderato (nel nostro caso 94 per avere il codice fiscale).

Il valore di offset, della lunghezza di 16 bit, è contenuto nella metà più bassa (seconda metà) della word che segue il codice operativo. Perciò, lo spostamento rispetto all'indirizzo di base, contenuto nel registro indirizzi può essere di più o meno 32Kbyte. L'offset è considerato come un numero in complemento a due; quindi, se il bit 15 è uno sarà considerato un valore negativo.

Sintassi dell'indirizzamento indiretto con offset

Nel formato standard dell'assembaltore dell'MC68000 l'indirizzamento indiretto a registro con offset è indicato facendo precedere il nome del registro indirizzi, racchiuso fra parentesi, da una label o da un valore immediato che rappresenta l'offset. Ad esempio, l'assemblatore trasforma lo statement

```
ADD $56(A3),D3
```

in un'istruzione ADD che somma il contenuto dell'indirizzo presente nel registro A3 più 56_{16} al contenuto del registro D3. Se il valore di spostamento fosse 8000_{16}, o più grande, sarebbe considerato come un valore negativo.

La Figura 3-11 mostra l'esecuzione di questa istruzione, che utilizza l'indirizzamento indiretto a registro con offset.

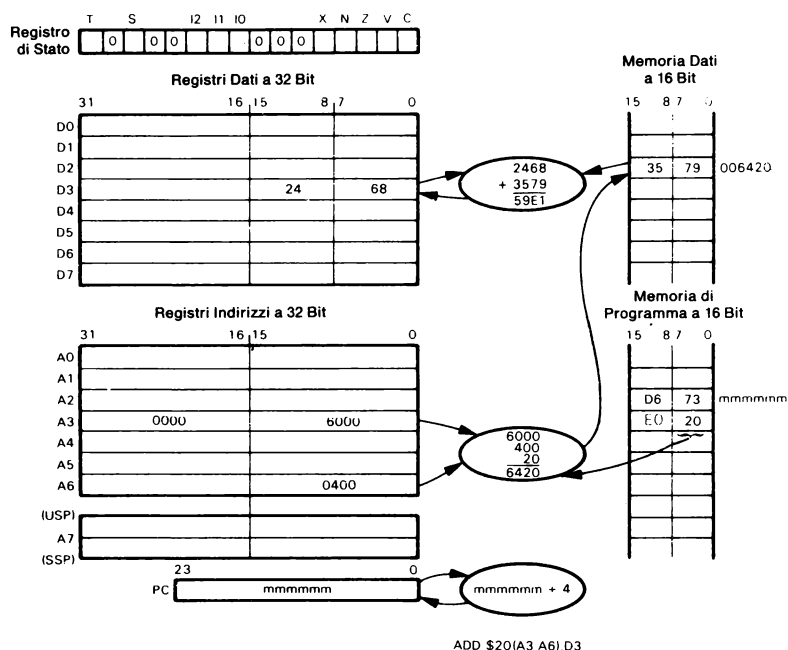
INDIRETTO A REGISTRO INDIRIZZI CON INDICE E SPOSTAMENTO

Questa volta l'indirizzo effettivo è la somma di tre indirizzi: il contenuto di un registro indirizzi, il contenuto di un registro indice, che può essere uno qualunque dei registri dati o indirizzi, ed un valore di spostamento, fornito dal codice oggetto dell'istruzione. Viene comunemente usato per accedere a dati strutturati. Ad esempio, se il registro indirizzi contiene l'inizio di un array di record, il registro indice viene usato per selezionare un particolare record: conterrà la distanza di quel particolare record dall'inizio dell'array. Poi, mediante l'offset è possibile selezionare un particolare campo di quello stesso record.

Il formato standard è molto simile a quello usato per l'indirizzamento indiretto a registro con offset. L'indicazione del registro è preceduta dal valore di spostamento. Il registro indirizzi ed il registro che deve essere usato come indice sono messi fra parentesi: prima viene specificato il registro indirizzi, seguito da una virgola, poi il registro indice. Ad esempio, l'assemblatore trasforma lo statement

```
ADD $20(A3,A6),D3
```

in una istruzione ADD che somma il contenuto di un indirizzo, ottenuto sommando i valori presenti nei registri A3 e A6 e l'offset

Figura 3-12. Indirizzamento Indiretto a Registro Indirizzi con Indice e Spostamento

20_{16}, al contenuto del registro D3. Il registro A3 rappresenta, nel nostro esempio, il registro base, A6 è il registro indice e l'offset (20_{16}) si trova nel byte di ordine basso della word, che segue il codice oggetto dell'istruzione nella memoria di programma.

La Figura 3-12 mostra l'esecuzione di questa istruzione ADD che si serve dell'indirizzamento indiretto a registro indirizzi con indice e offset.

Dimensioni dell'indice

Il registro indice può essere uno qualunque dei registri dati o indirizzi. È possibile specificare se l'indice è costituito solo dalla word di ordine basso del registro indice, con estensione del segno, come è illustrato nella Figura 3-12, oppure se come indice deve essere usato l'intero contenuto del registro (32 bit). L'indicazione della lunghezza dell'indice è contenuta nel byte di ordine alto della word, che segue il codice operativo dell'istruzione nella memoria di programma. È possibile indicare all'assemblatore di usare tutta la long word del registro indice aggiungendo un punto seguito dalla lettera L al nome del registro indice. Ad esempio, se volessimo usare l'intero contenuto del registro A6 come indice nella nostra istruzione ADD, il formato standard per l'assemblatore dell'MC68000 sarebbe

```
ADD $20(A3,A6.L),D3
```

Nel nostro esempio di indirizzamento indicizzato, non avevamo indicato la lunghezza dell'indice e, quindi, l'assemblatore presupponeva una lunghezza di una word, che è l'opzione di default (.W).

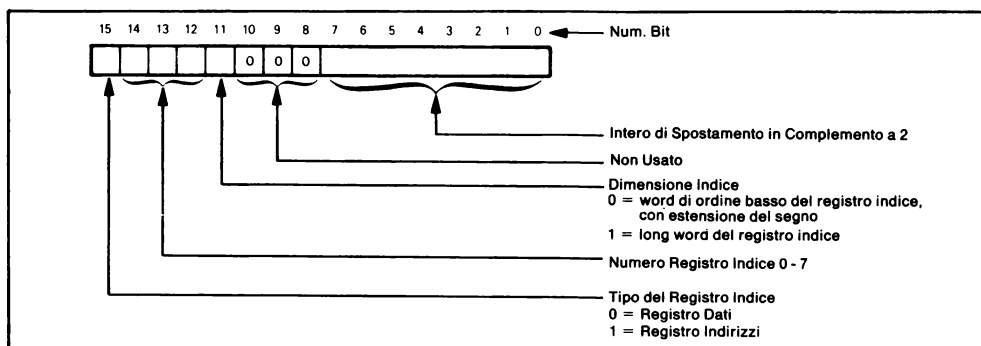



Figura 3-13. Assegnazione del Bit della Word di Estensione nell'indirizzamento Indiretto a Registro Indirizzi con Indice e Spostamento

Il formato del codice oggetto nella word di estensione per l'indirizzamento indiretto a registro indicizzato è descritto nella Figura 3-13. Il byte di ordine basso è l'intero che indica lo spostamento e viene considerato come un numero in complemento a due, fornito di segno, per cui sono possibili spostamenti compresi tra +127 e −128. Il bit 11 indica la lunghezza dell'indice. I bit da 12 a 14 specificano il numero del registro che fornisce l'indice e il bit 15 indica se è un registro dati o un registro indirizzi ad essere usato come registro indice.

INDIRIZZAMENTO RELATIVO AL CONTATORE DI PROGRAMMA

I tipi di indirizzamento che utilizzano uno spostamento rispetto al contatore di programma ci aiutano a scrivere un codice indipendente dalla posizione in memoria del programma, cioè programmi che sono in grado di funzionare in un'area qualsiasi della memoria. Programmi di questo tipo possono essere spostati, senza dover introdurre modifiche, in una qualunque zona della RAM disponibile e possono essere usati in combinazione con altri programmi. Il modo più semplice per realizzare un programma che abbia questa caratteristica è di specificare gli indirizzi che usa relativamente alla sua posizione. Come fa un programma a conoscere quali locazioni occupa? Usando il contentuo del contatore di programma.

Possiamo spostare un intero programma, insieme con i suoi dati, se indichiamo gli indirizzi relativamente al contatore di programma. Si tratta di indicare la posizione di un certo dato come posta, ad esempio "venti locazioni da dove ci troviamo adesso", anzichè specificarne l'indirizzo esatto. Se, successivamente, siamo costretti a spostare il programma in una zona diversa della memoria, le posizioni relative occupate dai dati e dalle istruzioni rimangono le stesse, anche se cambiano gli indirizzi assoluti.

Relativo al Contatore di Programma con Spostamento

Sintassi dell'indirizzamento relativo al PC con offset

In questo caso, la costante di spostamento o offset, rispetto al contatore di programma (PC) è fornita dal codice oggetto dell'istruzione. Il formato standard dell'assemblatore dell'MC68000 per questo tipo di indirizzamento è fondamentalmente lo stesso di quello usato per il modo indiretto a registro indirizzi con spostamento. È richiesto lo stesso formato, **perchè il modo relativo al contatore di programma è semplicemente un caso speciale dell'indirizzamento indiretto a registro;** in questo caso, il registro usato deve essere il contatore di programma, invece di uno qualsiasi dei registri indirizzi. L'assemblatore converte lo statement

```
ADD    $6(PC),D3
```

in un'istruzione che somma il contenuto della word di memoria posta sei byte oltre la locazione dell'istruzione al contenuto del registro D3.

La Figura 3-14 mostra l'esecuzione di questa istruzione ADD che impiega l'indirizzamento relativo al contatore di programma con spostamento. Noterete che in questo schema lo spostamento è applicato non al valore presente nel contatore di programma quando viene prelevato il codice oggetto dell'istruzione (mmmmmm), ma al valore presente nel contatore di programma dopo che questo è stato incrementato e punta alla word contenente il valore di spostamento.

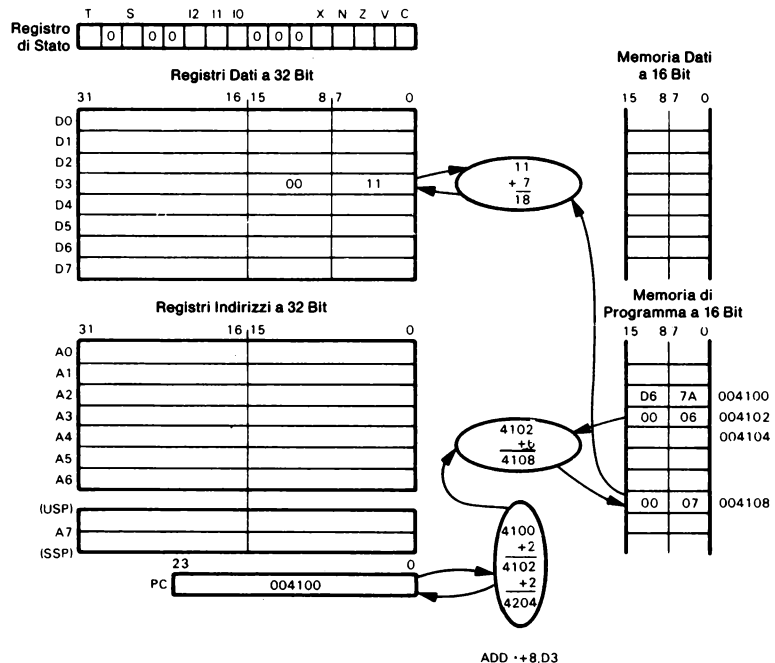



Figura 3-14. Indirizzamento Relativo al Contatore di Programma con Spostamento

L'assemblatore farà automaticamente questa correzione se utilizziamo il simbolo del contatore di locazione (*). In genere, tuttavia, ci serviremo di una label per specificare la costante di spostamento e, allora, l'assemblatore provvederà a generare il valore assoluto appropriato.

Relativo al Contatore di Programma con Indice e Spostamento

Questo tipo di indirizzamento è analogo a quello indiretto a registro indirizzi con indice e spostamento, tranne per il fatto che è usato il contatore di programma come registro base, invece di un registro indirizzi qualsiasi. Il formato standard per questo tipo di indirizzamento è sostanzialmente identico a quello indiretto a registro indicizzato. Ad esempio, l'assemblatore trasforma lo statement

ADD RSYMBOL(A3),D3

in una istruzione che somma al contenuto del registro D3 il contenuto della cella di memoria che ha l'indirizzo ottenuto sommando al contenuto del contatore di programma quello del registro A3 ed il valore di spostamento RSYMBOL. In questo esempio, il contatore di programma è il registro base, A3 il registro indice e lo spostamento è il valore ottenuto, con estensione del segno, dal byte di ordine basso successivo al codice d'istruzione.

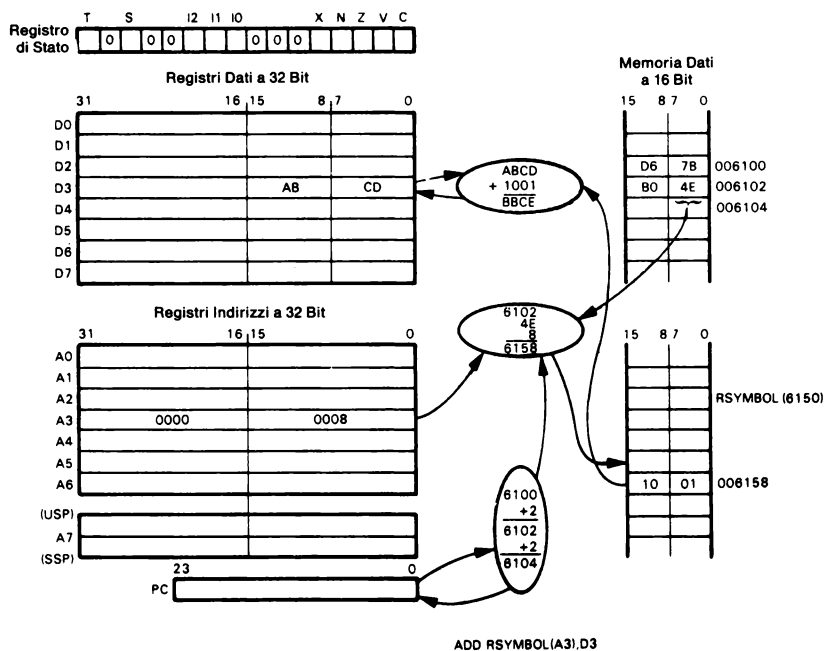



Figura 3-15. Indirizzamento Relativo al Contatore di Programma con Indice e Spostamento

Il valore di spostamento deve essere definito usando un simbolo relativo (anzichè uno assoluto), poichè questa forma indica implicitamente il contatore di programma come registro base. Un formato più comprensibile per questo tipo di indirizzamento sarebbe

```
ADD    $4E(PC,A3),D3
```

ed alcuni assemblatori lo accettano. Consultate il manuale del vostro assemblatore per stabilire con certezza i formati consentiti.

La Figura 3-15 mostra l'esecuzione di questa istruzione ADD che usa l'indirizzamento relativo al contatore di programma con indice e spostamento.

Il registro indice è uno qualsiasi dei registri dati o indirizzi, del quale può essere usata la word di ordine basso, con estensione del segno, come è indicato nella Figura 3-15, oppure l'intero valore a 32 bit. L'indicazione relativa alla lunghezza dell'indice è contenuta nel byte di ordine alto della parola, che segue nella memoria di programma il codice operativo dell'istruzione. Il formato del codice oggetto per la word di estensione, previsto dall'indirizzamento relativo al contatore di programma con indice e spostamento, è identico a quello usato per l'indirizzamento indiretto a registro indirizzi con indice e spostamento (cfr. la Figura 3-13).

INDICAZIONE DEL MODO DI INDIRIZZAMENTO

Per molte istruzioni non c'è la possibilità di specificare il tipo di indirizzamento: ad esempio, le istruzioni Exchange usano sempre operandi contenuti nei registri. In altre istruzioni, **ad esempio RESET o NOP,** l'indicazione dell'indirizzamento non è necessaria, data la mancanza di un operando. **Per quelle istruzioni per cui è possibile specificare il tipo di indirizzamento da usare sono utilizzati, a questo scopo, i sei bit meno significativi della word del codice operativo dell'istruzione.** I bit 0, 1 e 2 indicano, in alcuni tipi di indirizzamento, il numero di un registro e, in altri casi, sono usati, unitamente ai bit 3, 4 e 5, per specificare ulteriormente il modo di indirizzamento.

La Tabella 3-4 mostra come questi sei bit determinano quale indirizzamento deve essere usato. Come potete notare nei modi da 000 a 110 i tre bit meno significativi indicano il numero di un registro, poichè viene sempre utilizzato un registro dati o un registro indirizzi per ottenere l'indirizzo effettivo. Nei casi restanti i bit di modo sono tutti posti a 1 ed i tre bit meno significativi del codice operativo non indicano più il numero di un registro, ma selezionano, invece, uno dei modi di indirizzamento che non usano un registro dati o indirizzi come registro base o primario. I testi della Motorola classificano questi indirizzamenti come speciali, ma la sola cosa che hanno di "speciale" è il fatto che non usano i tre bit meno significativi per indicare il numero di un registro.

Tabella 3-4 - Indicazione del Modo di Indirizzamento

Tipo di Indirizzamento	Modo	Num. Reg.
Num. Bit →	5 4 3	2 1 0
Diretto a Registro Dati	0 0 0	r r r
Diretto a Registro Indirizzi	0 0 1	r r r
Indiretto a Registro indirizzi	0 1 0	r r r
Indiretto a Registro con Postincremento	0 1 1	r r r
Indiretto a Registro Indirizzi con Predecremento	1 0 0	r r r
Indiretto a Registro Indirizzi con Spostamento	1 0 1	r r r
Indiretto a Registro Indirizzi con Indice e Spostamento	1 1 0	r r r
Assoluto Corto	1 1 1	0 0 0
Assoluto Lungo	1 1 1	0 0 1
Relativo al Contatore di Programma con Spostamento	1 1 1	0 1 0
Relativo al Contatore di Programma con Indice e Spostamento	1 1 1	0 1 1
Immediato o a Registro di Stato	1 1 1	1 0 0

CONVENZIONI DELL'ASSEMBLATORE MOTOROLA PER L'MC68000

L'assemblatore standard per l'MC68000 è disponibile presso la Motorola e le più importanti reti a time-sharing, oltre ad essere incluso in molti sistemi di sviluppo. Versioni di tipo cross-assembler sono disponibili anche per gran parte dei grossi sistemi e dei mini-computer. L'assemblatore residente differisce, per alcuni aspetti, dai vari cross-assembler. È opportuno consultare il relativo manuale.

DELIMITATORI DEI CAMPI

Le istruzioni in linguaggio assembly hanno una struttura standard costituita da vari campi (cfr. la Tabella 2-1). I delimitatori richiesti sono:

1. **Uno spazio o i due punti dopo una label.** Un'etichetta che inizia nella colonna 1 deve essere seguita da almeno uno spazio, mentre una che inizia in una colonna diversa deve terminare con i due punti. Tutte le istruzioni non provviste di label devono essere precedute da uno spazio.
2. **Uno spazio dopo il codice operativo.** Quelle istruzioni in grado di operare su dati di lunghezza diversa possono avere un codice indicante la lunghezza del dato aggiunto al codice operativo senza uno spazio. Questo codice è rappresentato da un punto (.) seguito da B, W o L (B = byte, W = word, L = long word). Ad esempio, ADD.L sta per "add long word" (operando di 32 bit) e ADD.B per "add byte" (operando di 8 bit). Se non è specificata la dimensione del dato, l'assemblatore utilizza il valore di default, che è quello di una word.
3. **Una virgola fra gli operandi nel campo indirizzi:** ad esempio, tra un valore immediato ed un registro. ADD #5,D1 somma il valore 5 al registro D1.

4. **Le parentesi - ( ) - racchiudono il nome di un registro, quando questo viene usato in modo indiretto per ottenere un indirizzo.**
5. **Un segno più dopo le parentesi per indicare l'indirizzamento con postincremento; un segno meno prima delle parentesi per indicare l'indirizzamento con predecremento.**
6. **Uno spazio prima di un commento che si trova sulla stessa riga di un'istruzione ed un asterisco prima di una riga destinata al solo commento.**

LABEL

Dimensione di una label

Composizione di una label

Con l'assemblatore della Motorola solo i primi otto caratteri di una label sono significativi. Ulteriori caratteri sono ignorati dall'assemblatore, benchè vengano ugualmente stampati nel listato del programma. Quindi, l'assemblatore non farà differenza fra LABELNUMBER1 e LABELNUMBER2: entrambe saranno considerate come una sola etichetta, LABELNUM. Il primo carattere deve essere una lettera maiuscola o il carattere speciale punto (.). Gli altri caratteri possono essere una lettera, una cifra (0-9), il segno del dollaro, un punto, una lineetta oppure un segno di sottolineatura. Benchè il formato standard preveda, in alcuni casi, l'uso di mnemonici operativi come label, generalmente questa non è una buona abitudine, per la confusione che ne può derivare.

DIRETTIVE DELL'ASSEMBLATORE

L'assemblatore ha le seguenti psueudo-operazioni esplicite:

ORG	- Pone (il contatore di locazione al)l'origine
SECTION	- Stabilisce il valore del contatore di programma per una sezione del programma che è rilocabile
END	- Fine del programma sorgente
EQU	- Definisce un nome simbolico permanente
SET	- Definisce un nome simbolico temporaneo
REG	- Definisce un elenco di registri
DC	- Definisce dati costanti
DCB	- Definisce blocchi di dati costanti
DS	- Definisce uno spazio di memoria destinato a contenere dei dati

DC e DCB - Direttive Dati

DC e DCB sono le direttive dati usate per mettere delle costanti nella memoria di programma: si tratta di tabelle, messaggi e fattori numerici, necessari per l'esecuzione del programma, ma che non fanno parte delle istruzioni. DC è usata per definire costanti decimali, esadecimali o ASCII. Le costanti definite possono essere

di un byte, di una word e di una long word. È possibile definirne la lunghezza aggiungendo B (byte), W (word) o L (long word). Una singola direttiva DC è sufficiente per definire costanti multiple, separate fra loro da una virgola.

Esempi:

```
DC.B 10,5,7
```

mette i numeri decimali 10, 5 e 7 in tre byte consecutivi della memoria. Se viene definito un numero dispari di byte, viene posto uno zero nel byte dispari, a meno che non segua un'altra DC.B.

```
DC.B   'ERROR'
```

mette i corrispondenti codici ASCII a 7 bit di E, R, R, O e R (i valori esadecimali 45, 52, 52, 4F e 52) nei cinque byte successivi della memoria di programma. La stringa ASCII deve essere racchiusa tra due apostrofi (').

```
DC.L   10,5,7
```

In questo modo vengono definite tre long word consecutive della memoria. Il valore 10 occupa i primi quattro bit meno significativi della long word, mentre la parte restante (quella più significativa) è occupata da zeri. Il valore 5 è nella seconda long word ed il 7 nella terza.

DCB - Define Constant Block

È la direttiva usata per inizializzare con un determinato valore un blocco di byte, di word o di long word (a seconda del codice che segue la direttiva). Esempio:

```
DCB.B 5,0
```

mette uno zero in cinque byte consecutivi della memoria. Questa direttiva è impiegata raramente all'interno della logica di un programma e serve soprattutto a definire dei valori iniziali in alcune locazioni di memoria (come tutti zero o tutti uno).

DS - Define storage

DS è la direttiva usata per riservare delle locazioni di memoria destinate a scopi particolari. Riserva un determinato numero di byte (.B), di word (.W) o di long word (.L), a seconda della specifica di lunghezza. La memoria riservata non viene inizializzata in alcun modo.

EQU - Equate

EQU è la direttiva usata per definire dei nomi.

SET

Serve a definire dei nomi ed è, quindi, simile alla direttiva precedente. La differenza sta nel fatto che la direttiva SET **permette di ridefinire un simbolo in una parte successiva del programma, mediante una nuova direttiva dello stesso tipo.**

ORG

È la direttiva standard di origine. I programmi in linguaggio assembly per l'MC68000 hanno, di solito, diverse origini, che sono impiegate per i seguenti scopi:

1. Indicare l'indirizzo iniziale del programma principale
2. Indicare l'indirizzo iniziale di una subroutine
3. Definire aree di memoria per immagazzinare dati
4. Definire aree di memoria come stack per l'utente
5. Indicare indirizzi usati come porte di I/O e per funzioni speciali

SECTION

La direttiva SECTION è simile a quella di origine per il fatto che stabilisce i punti di partenza dei programmi e delle subroutine. **Serve a creare sezioni rilocabili di programma,** per cui, tramite un linker, è possibile ottenere successivamente dei programmi eseguibili.

END

END indica semplicemente la fine di un programma in linguaggio assembly.

INDIRIZZI

L'assemblatore della Motorola consente di introdurre dei valori nel campo indirizzi in uno dei seguenti formati:

1. Decimale (l'opzione di default)
   Esempio: 12345
2. Esadecimale (deve iniziare con $)
   Esempio: $CE00

3 **Ottale (deve iniziare con un #)**
Esempio: #1247
4. Binario (deve iniziare con %)
Esempio: %00101
5 ASCII (deve essere racchiuso tra due apostrofi)
Esempio: 'ABCD'
6. Come offset rispetto al valore attuale del contatore di locazione (asterisco)
Esempio: *+10

Espressioni Logiche ed Aritmetiche

L'assemblatore permette anche di inserire in un campo indirizzi delle espressioni costituite da nomi e numeri separati da operatori aritmetici:

+ addizione
– sottrazione
* moltiplicazione
/ divisione
> > shift a destra
< < shift a sinistra
& AND logico
! OR logico

L'ordine di precedenza dei vari operatori è il seguente:

1. Le espressioni fra parentesi sono valutate per prime.
2. Poi è il turno delle operazioni di shift.
3. Le operazioni logiche di AND e di OR hanno la priorità successiva.
4. Gli operatori di moltiplicazione hanno la precedenza su quelli di addizione e sottrazione.
5. Operatori con la stessa precedenza sono valutati da sinistra verso destra.

Tutti i risultati, compresi quelli intermedi, sono troncati a interi a 32 bit.

Raccomandiamo di evitare l'uso di espressioni nel campo indirizzi tutte le volte che è possibile, dal momento che non esistono degli standard riguardo alle modalità di calcolo. Se non potete farne a meno, allora commentate tutte le espressioni non troppo chiare e assicuratevi che la loro valutazione non produca mai dei risultati troppo grandi per l'uso che dovete farne.

ALTRE CARATTERISTICHE DELL'ASSEMBALTORE

La maggior parte degli assemblatori hanno altre caratteristiche, fra cui la possibilità di impiegare delle macro e l'assemblaggio condizionato. Altre ancora (come la direttiva SECTION) sono strettamente correlate all'uso del linker e, per una descrizione completa, consigliamo di consultare i testi forniti dalla Motorola. Per avere maggiori dettagli sul modo in cui i diversi assemblatori svolgono queste funzioni, è opportuno leggere i relativi manuali.

IL SET DI ISTRUZIONI DELL'MC68000

La Tabella 3-5 elenca i mnemonici d'istruzione dell'MC68000. Per una descrizione dell'intero set d'istruzioni rimandiamo all'ultima parte di questo volume. Nel Capitolo 22 analizzeremo dettagliatamente ogni singola istruzione e vi consigliamo di consultarlo, ogni volta che avrete necessità di sapere come funziona una determinata istruzione. Nell'Appendice A troverete un sommario di tutte le istruzioni dell'MC68000, raggruppate in base alla loro funzione. Questo fornisce un quadro complessivo delle capacità dell'MC68000 e vi sarà utile anche quando, dovendo eseguire un certo tipo di operazione, non siete sicuri del mnemonico richiesto o non avete ancora molta

Tabella 3-5 - Mnemonici d'istruzione

Mnemonico	Descrizione	Mnemonico	Descrizione
ABCD	Somma decimale con riporto	MOVE	Trasferimento
ADD	Somma	MOVEM	Trasferimento tra registri e memoria
AND	AND logico	MOVEP	Trasferimento tra processore e periferiche
ASL	Shift aritmetico a sinistra	MULS	Moltiplicazione tra numeri con segno
ASR	Shift aritmentico a destra	MULU	Moltiplicazione tra numeri senza segno
B_{cc}	Diramazione (Branch) condizionata	NBCD	Negazione di un decimale
BCHG	Test su un bit e cambio del suo valore	NEG	Negazione
BCLR	Testa un bit e lo pone a 0	NOP	Nessuna operazione
BRA	Diramazione incondizionata	NOT	Complemento a 1
BSET	Testa un bit e lo pone a 1	OR	OR logico
BSR	Diramazione a una subroutine	PEA	Caricamento di un indirizzo nello stack
BTST	Testa lo stato di un bit	RESET	Reset di un device
CHK	Controllo del valore di un registro	ROL	Rotazione a sinistra
CMP	Confronto	ROR	Rotazione a destra
DB_{cc}	Test, decremento e diramazione	ROXL	Rotazione a sinistra con riparto
DIVS	Divisione tra numeri con segno	ROXR	Rotazione a destra con riparto
DIVU	Divisione tra numeri senza segno	RTE	Ritorno da un'exception
EOR	OR chiuso	RTR	Ritorno a Restore
EXG	Scambio del contenuto di 2 registri	RTS	Ritorno da una subroutine
EXT	Estensione del bit di segno	SBCD	Sottrazione decimale con riporto
JMP	Salto	S_{cc}	Riempie di 1 un registro se la condizione è vera
JSR	Salto a subroutine	STOP	Stop
LEA	Caricamento di un indirizzo	SUB	Sottrazione
LINK	Crea un link con lo stack	SWAP	Scambio di bit in un registro
LSL	Shift logico a sinistra	TAS	Test and Set
LSR	Shift logico a destra	TRAP	Inizia il trattamento di una exception
		TRAPV	Trattamento di un overflow
		TST	Test
		UNLK	Toglie il link con lo stack

familiarità con le istruzioni disponibili. Le appendici restanti servono come tabelle di riferimento per il calcolo del tempo di esecuzione di un programma e della memoria necessaria e per un eventuale assemblaggio o disassemblaggio manuale.

Le istruzioni spesso intimidiscono coloro che utilizzano un microcomputer ma non hanno esperienza di programmazione. Eppure, di solito, le operazioni connesse all'esecuzione di una singola istruzione, se prese singolarmente, sono facili da capire. Lo scopo dell'ultima parte di questo libro è di spiegare, una per una, queste operazioni. Inoltre non cercate di imparare tutte le istruzioni in una volta. Man mano che studierete i programmi presentati in questo libro, imparerete anche a cosa servono le istruzioni che vengono di volta in volta utilizzate.

Perchè ci riferiamo alle istruzioni di un microprocessore come *set d'istruzioni?* Perchè il progettista di un microprocessore sceglie le istruzioni con una cura particolare, in modo che sia possibile eseguire delle funzioni complesse come una sequenza di semplici fasi, ciascuna delle quali è rappresentata da una istruzione appartenente ad un set ben progettato.

SEZIONE II

PROBLEMI INTRODUTTIVI

L'unico modo per imparare a programmare in linguaggio assembly è attraverso l'esperienza. I prossimi sei capitoli di questo libro contengono semplici esempi di programmazione, che svolgono alcune delle funzioni normalmente affidate ad un microprocessore. Consigliamo di leggere attentamente ciascun programma e di eseguirlo su un microcomputer dotato dell'MC68000. Quindi, esercitatevi con i problemi che sono alla fine di ogni capitolo, per assicurarvi di averne compreso a fondo il contenuto.

Gli esempi vi forniscono le indicazioni necessarie alla soluzione dei problemi. Non dimenticate di eseguire i vostri programmi su un microcomputer dotato di un MC68000, per essere sicuri che siano corretti.

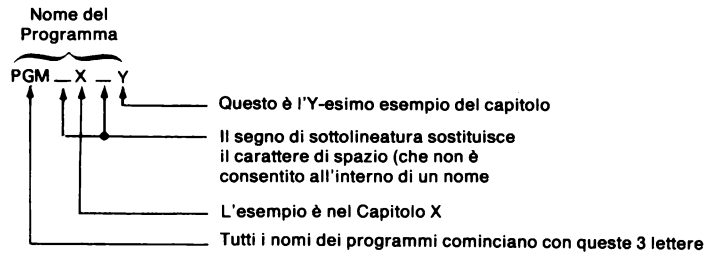
FORMATO GENERALE DEGLI ESEMPI

Ciascun esempio è formato da:

- **Un titolo** che fornisce indicazioni sul tipo di problema
- **Una descrizione dello scopo** che descrive la funzione svolta dal programma e le locazioni di memoria impiegate
- **Un problema d'esempio** con i dati ed i risultati
- **Un diagramma di flusso** se la logica del programma è particolarmente complessa
- **Il programma sorgente,** cioè listato in linguaggio assembly
- **Il programma oggetto,** cioè il listato esadecimale in linguaggio macchina
- **Note esplicative** che descrivono le istruzioni ed i metodi utilizzati nel programma

Ogni esempio è scritto ed assemblato come un programma a sè stante. A ciascuno è stato assegnato un nome che identifica il

capitolo in cui si trova e la posizione sequenziale occupata all'interno del capitolo, nel modo seguente:

Schema pag. II-2 in alto

X e N rappresentano due numeri decimali qualsiasi. Perciò, con

PGM_6_3

si intederà il terzo esempio del Capitolo 6.

Formato del Listato

I listati mostreranno il codice oggetto unitamente al programma sorgente: non è altro che il formato di output di un comune assemblatore. Ecco, ad esempio, una parte del programma 4-1:

```
00006000:                DATA      EQU      $6000
00004000:                PROGRAM   EQU      $4000
                         ;
                                   ORG      DATA
00006000:                VALUE     DS.W     1            VALORE DA TRASFERIRE
00006002:                RESULT    DS.W     1            LOC. DOVE SALVARE IL DATO
                         ;
                                   ORG      PROGRAM
                         ;
00004000: 3038 6000      PGM_4_1   MOVE.W   VALUE,D0     PRENDI IL DATO
00004004: 31C0 6002                MOVE.W   D0,RESULT    SALVA IL DATO
                         ;
00004008: 4E75                     RTS
                                   END      PGM_4_1
```

Il numero a sei cifre, che inizia nella colonna più a sinistra di ogni riga, è l'indirizzo esadecimale del primo byte del codice oggetto presente su quella riga. 004000 è l'indirizzo del primo byte del codice oggetto dell'istruzione MOVE.W VALUE,D0, formata da quattro byte; il codice oggetto esadecimale è 3038 6000 ed il primo byte, 30, si trova nella locazione 4000. La locazione 4001 contiene il byte 38 - lo si intuisce dal fatto che 38 segue il byte dell'indirizzo 4000 - e le locazioni 4002 e 4003 contengono, rispettivamente, i byte 60 e 00. Le parole a destra del codice oggetto sono i campi del linguaggio assembly, come sono stati descritti nel Capitolo 2.

Se desiderate assemblare questi esempi sul vostro microcomputer, battete solamente le istruzioni del programma sorgente, tralasciando indirizzi e codici oggetto, in quanto è l'assemblatore che provvederà a generarli. Sarà necessario anche inserire alcune direttive rivolte all'assemblatore (ad esempio, indicare all'assemblatore quale deve essere l'indirizzo iniziale). Vi mostreremo alcune di queste direttive, ma quelle che dovrete usare dipenderanno dal tipo di assemblatore adottato e dal sistema operativo del vostro elaboratore.

Chi desidera eseguire gli esempi senza dover assemblare il programma sorgente, può digitare direttamente il codice oggetto agli indirizzi indicati. Prima, però, è opportuno accertarsi di non utilizzare aree riservate al monitor o al sistema operativo. Per evitare problemi di questo tipo, può essere necessario cambiare gli indirizzi, prima di caricare il programma. Come vedremo qui di seguito, al punto 7, in alcuni casi sarà indispensabile cambiare anche l'istruzione che si trova alla fine del programma.

Caratteristiche degli Esempi

Nella realizzazione degli esempi, abbiamo seguito le seguenti modalità:

1. È stata impiegata la notazione standard dell'assemblatore Motorola, descritta nel Capitolo 3.
2. La forma in cui appaiono gli indirizzi e i dati è stata scelta tenendo presente la chiarezza, piuttosto che l'uniformità. Sono stati usati numeri esadecimali per gli indirizzi, i codici d'istruzione e i dati BCD; decimali per le costanti numeriche; binari per le maschere logiche; ASCII per i caratteri.
3. Si è cercato di sottolineare le istruzioni e le tecniche di programmazione usate più frequentemente.
4. Gli esempi mostrano le funzioni che un microprocessore è in grado di svolgere quando viene utilizzato all'interno di un elaboratore o di altre apparecchiature, nel campo delle comunicazoni o in applicazioni di tipo commerciale, industriale o militare.
5. Sono stati inseriti dei commenti molto dettagliati.
6. I programmi sono stati scritti in una forma tale da garantire la massima efficienza possibile e, al tempo stesso, la massima chiarezza e semplicità strutturale. Nelle note che li accompagnano sono descritte, in alcuni casi, delle procedure alternative più efficienti.
7. I programmi mostrano una certa uniformità nella utilizzazione della memoria. Ciascun programma inizia alla locazione 4000_{16} e termina con un'istruzione RTS (Return from Subroutine); è scritto, inoltre, come una procedura o subroutine indipendente, benchè venga completamente trascurato lo stato del microprocessore al momento in cui inizia la sua esecuzione. È possibile,

per chi lo desidera, modificare il modo in cui termina un programma, sostituendo, ad esempio, un'istruzione RTS con un'istruzione STOP o con un ciclo infinito del tipo:

```
HERE: JMP HERE
```

8. I programmi usano le direttive standard proprie dell'assemblatore Motorola, descritte nel Capitolo 2. Esaminando per la prima volta gli esempi, potete benissimo trascurare le direttive dell'assemblatore, se non riuscite a capirle. Non servono a chiarire la logica del programma (che è la prima cosa che bisogna capire), ma sono una componente indispensabile di ogni programma in linguaggio assembly: dovrete, quindi, imparare ad usarle prima di scrivere voi stessi dei programmi. La presenza di queste direttive in tutti gli esempi vi aiuterà ad acquistare una certa familiarità con le loro funzioni.

Provare gli Esempi

Per provare un esempio sul vostro computer è necessario innanzitutto introdurre in memoria il programma oggetto. Alcuni assemblatori lo fanno automaticamente, mentre altri creano un file con il codice oggetto che un caricatore provvede, poi, a mettere in memoria.

Nelle maggior parte degli esempi, le locazioni usate per i dati sono state specificate usando la direttiva DS (Define Storage), che stabilisce, semplicemente, le locazioni di memoria che devono essere riservate e che saranno, poi, utilizzate dal programma; a differenza della direttiva DC (Define Constant), essa non assegna alcun valore iniziale alle locazioni di memoria riservate. La direttiva DS viene utilizzata anche per indicare locazioni di memoria destinate a contenere indirizzi di variabili (ad es., l'indirizzo di un array o di una tabella). **Una volta che il programma è in memoria, bisogna mettere i dati (e/o gli indirizzi) nelle locazioni appropriate e, quindi, provare ad eseguirlo. Una volta terminato, vanno esaminate le locazioni contenenti i risultati.** Per provare il programma con dei dati diversi, è sufficiente cambiare i valori presenti nelle locazioni dei dati, prima di eseguirlo nuovamente. (Se il sistema utilizzato non consente di introdurre direttamente i dati in memoria, è necessario utilizzare le direttive DC).

L'uso delle locazioni di memoria per fornire al programma delle informazioni variabili è definito passaggio di parametri (i dati da elaborare e alcuni degli indirizzi da usare rappresentano i parametri). È una tecnica molto diffusa, impiegata insieme alle subroutine; ne parleremo più dettagliatamente nei Capitoli 10 e 11. A questo punto, tutto quello di cui dovete preoccuparvi, quando volete provare a far girare uno degli esempi, è di caricare le informazioni riguardanti i dati e gli indirizzi nelle aree di memoria prestabilite.

INIZIALIZZAZIONE

Tutti gli esempi presentati nel Capitolo 4, e nei capitoli successivi, prestano una particolare attenzione ad una corretta inizializzazione delle costanti e degli operandi. Questo richiede, spesso, delle istruzioni addizionali che possono apparire superflue, nel senso che non contribuiscono direttamente alla soluzione del problema descritto. Tuttavia, una corretta inizializzazione è importante per far sì che il programma venga ogni volta eseguito in modo corretto.

Ci preme sottolineare l'importanza di una corretta inizializzazone: per questo motivo poniamo l'accento su questo aspetto del problema.

CONDIZIONI SPECIALI

Per gli stessi motivi per cui attribuiamo grande importanza alla fase di inizializzazione, vogliamo sottolineare particolari condizioni che provocano un'errata esecuzione di un programma. Liste vuote e indici azzerati sono due delle circostanze più comuni analizzate in alcuni degli esempi. **È estremamente importante** nell'uso dei microprocessori in generale, e di quelli più potenti a 16 bit in particolare, **imparare, fin dall'inizio, a prevedere situazioni insolite, che spesso non consentono una corretta esecuzione. È indispensabile che una parte del programma si occupi di questi problemi potenziali.**

CONSIGLI PER LA SOLUZIONE DEI PROBLEMI

Per risolvere i problemi posti alla fine di ogni capitolo, è opportuno rispettare le seguenti modalità:

1. Commentare ogni programma, in modo che anche altri possano capirlo. Non importa se i commenti sono brevi e sgrammaticati. Devono spiegare lo scopo di un'istruzione o di un'intera sezione del programma, non quello che fa ogni singola istruzione, poichè questo lo troviamo già nei manuali. Non sforzatevi di chiarire quello che di per sè è già evidente. I commenti contenuti negli esempi possono servire come modello, anche se, per motivi di chiarezza, spesso sono eccessivamente prolissi.
2. La chiarezza, la semplicità e un buona struttura del programma devono rappresentare l'obiettivo principale. I programmi devono essere ragionevolmente efficienti, ma non preoccupatevi di risparmiare un byte di memoria o di ridurre di qualche microsecondo il tempo di esecuzione.
3. I programmi devono essere ragionevolmente generali. Non bisogna confondere i parametri (come il numero degli elementi di un vettore) con le costanti (come un carattere ASCII).

4. Non assegnare mai dei valori iniziali costanti ai parametri.
5. Usare la notazione propria dell'assemblatore, come mostrato negli esempi e descritto nel Capitolo 3.
6. Usare notazioni simboliche per i riferimenti agli indirizzi e ai dati. La notazione simbolica dovrebbe essere utilizzata anche per le costanti (come DATA SELECT invece di %00000100). Anche per i dati è meglio adottare la forma che risulta più chiara (come 'C' invece di $43).
7. Se il sistema lo consente, iniziare tutti i programmi alla locazione di memoria 4000_{16} e servirsi delle locazioni a partire da 6000_{16} per i dati e per memorizzare temporaneamente dei valori e della locazione 7000_{16} come base dello stack. Altrimenti è opportuno stabilire degli indirizzi equivalenti ed utilizzarli al posto di quelli appena indicati. Vi consigliamo, in ogni modo, di consultare il manuale del vostro sistema.
8. Utilizzare nomi significativi per le label e le variabili, ad es. SUM o CHECK piuttosto che X o Z.
9. Eseguire i programmi una volta che sono stati realizzati non c'è altro modo per essere sicuri che siano corretti. Per ogni problema troverete dei dati campione. Assicuratevi che il programma funzioni anche in presenza di condizioni particolari. I Capitoli 19 e 20 forniscono alcuni utili indicazioni da seguire durante il collaudo dei programmi.

CAPITOLO 4

PROGRAMMI SEMPLICI

Questo capitolo contiene dei programmi molto semplici, che serviranno ad introdurre alcune delle caratteristiche fondamentali dell'MC68000, oltre che a mostrare alcune funzioni di base comuni a programmi in linguaggio assembly destinati alle più diverse applicazioni.

ESEMPI DI PROGRAMMAZIONE

4-1. Trasferimento di un dato a 16 bit

Scopo: Trasferire il contenuto della variabile a 16 bit VALUE, posta alla locazione 6000, in un'altra variabile a 16 bit, RESULT, alla locazione 6002.

Problema Campione:

Input: VALUE-(6000) = 2E56
Output: RESULT-(6002) = 2E56

Programma 4-1:

```
00006000:              DATA     EQU     $6000
00004000:              PROGRAM  EQU     $4000
                       ;
                                ORG     DATA
00006000:              VALUE    DS.W    1            VALORE DA TRASFERIRE
00006002:              RESULT   DS.W    1            LOC. DOVE SALVARE IL DATO
                       ;
                                ORG     PROGRAM

                       ;
00004000: 3038 6000    PGM_4_1  MOVE.W  VALUE,D0     PRENDI IL DATO
00004004: 31C0 6002             MOVE.W  D0,RESULT    SALVA IL DATO
                       ;
00004008: 4E75                  RTS

                                END     PGM_4_1
```

Spiegazione del programma 4.1

Questo programma risolve il problema in due fasi successive. La prima istruzione carica nel registro dati D0 il valore a 16 bit prelevato dalla locazione VALUE. L'istruzione seguente salva il contenuto a 16 bit del registro dati D0 nella locazione RESULT.

Se volete provare questo programma con qualche dato campione, dovete prima caricare il dato da trasferire nella variabile VALUE, alla locazione di memoria 6000. Qualora il vostro sistema non lo consenta, dovrete ricorrere alla direttiva Define Constant.

Nell'esecuzione del programma sono coinvolti solamente i 16 bit meno significativi del registro dati D0. I 16 bit più significativi sono ignorati, in quanto entrambe le istruzioni specificano un operando della grandezza di una word (16 bit) mediante il suffisso '.W'. Volendo trasferire un dato della grandezza di un byte (8 bit) o di una long word (32 bit), è necessario usare, rispettivamente, i suffissi '.B' e '.W'.

L'istruzione MOVE

L'MC68000 combina fra loro tre istruzioni (carica un registro, salva un registro e trasferisci da un registro all'altro), che nella maggior parte dei microprocessori sono distinte, in un'istruzione unica: MOVE. L'uso di un registro come operando sorgente (primo operando) in un'istruzione MOVE, equivale a salvare un registro in un normale microprocessore. L'uso di un registro come operando destinazione equivale, in un normale microprocessore, a caricare un determinato valore in un registro. Infine, l'uso dei registri interni come operandi sorgente e destinazione in un'istruzione MOVE corrisponde ad un'istruzione di trasferimento fra registri di un microprocessore tipico.

L'impiego di un'istruzione MOVE per svolgere le funzioni LOAD, STORE o TRANSFER, modifica generalmente i flag del registro di stato. L'esecuzione di gran parte delle istruzioni MOVE pone a 1 o azzera i flag di Negativo (N) e di Zero (Z), a seconda del valore trasferito, azzerando, allo stesso tempo, l'Overflow (V) ed il Carry (C). Il flag di Extend (E) non viene modificato.

Uso dell'istruzione MOVE per spostamenti tra celle di memoria

Oltre a trasferire dei dati tra un registro e l'altro e tra i registri e la memoria, l'istruzione MOVE può essere usata anche per spostare dati tra due locazioni di memoria. Ne deriva che potremmo sostituire le due istruzioni MOVE del PGM 4-1 con un'unica istruzione:

```
MOVE.W    VALUE,RESULT
```

Questa versione dell'istruzione MOVE trasferisce la word di 16 bit, contenuta nella locazione di memoria VALUE, nella locazione RESULT, senza utilizzare nessuno dei registri dati o indirizzi. Il registro di stato viene modificato anche in questo caso.

Esaminando il set d'istruzioni dell'MC68000 si noterà che sono molte le istruzioni in grado di operare sulla memoria in modo analogo.

4-2. Complemento a uno

Scopo: Formare il complemento, bit a bit, del contenuto della variabile a 16 bit VALUE, alla locazione 6000.

Problema Campione:

Input: VALUE – (6000) = 7F3E
Output: RESULT – (6002) = 80C1

Programma 4-2:

```
00006000:                DATA    EQU     $6000
00004000:                PROGRAM EQU     $4000
                         ;
                                 ORG     DATA
00006000:                VALUE   DS.W    1          VALORE DA COMPLEMENTARE
                         ;
                                 ORG     PROGRAM
                         ;
00004000: 3038 6000      PGM_4_2 MOVE.W  VALUE,D0   PRENDI IL DATO
00004004: 4640                   NOT.W   D0         COMPLEMENTO LOGICO DEL VALORE
00004006: 31C0 6000              MOVE.W  D0,VALUE   SALVA IL RISULTATO COMPLEMENTATO
                         ;
0000400A: 4E75                   RTS

                                 END     PGM_4_2
```

Spiegazione del programma 4.2

Questo programma risolve il problema in tre fasi. La prima istruzione sposta il contenuto della locazione VALUE nel registro dati D0. Quella successiva esegue il complemento logico del registro dati D0. Infine, nella terza istruzione il risultato del complemento logico viene messo in VALUE.

È da notare che con una istruzione che usa i registri dati si può impiegare uno qualsiasi di questi registri. (La stessa cosa vale per i registri indirizzi, sebbene sia necessaria una particolare attenzione con il registro A7, di cui il processore si serve come puntatore allo stack). Quindi, nell'istruzione MOVE che abbiamo appena descritto, poteva essere utilizzato uno qualunque dei registri dati.

Le due istruzioni MOVE di questo programma, come quelle del Programma 4-1, mostrano due dei modi di indirizzamento dell'MC68000. Il riferimento a VALUE come operando sorgente o destinazione è un esempio di indirizzamento assoluto. Nell'indirizzamento assoluto l'indirizzo di un dato è contenuto nella (o nelle) word di estensione successive a quella del codice operativo. Come si può rilevare dal listato, l'indirizzo (6000) corrispondente a VALUE si trova nella word di estensione delle istruzioni MOVE.

Indirizzamento assoluto corto

Dal momento che l'indirizzo di VALUE richiede solo una word di estensione, l'MC68000 definisce questa forma di indirizzamento assoluto come indirizzamento assoluto corto. La forma corta è impiegata con indirizzi compresi tra 00000000 e 00007FFF e tra FFFF8000 e FFFFFFFF. Forse vi sareste aspettati dei valori diversi, ma questi sono dovuti al modo in cui l'MC68000 gestisce gli indirizzi a 16 bit e gli offset, che sono sempre valori a 32 bit provvisti di segno. Una tecnica di indirizzamento di questo tipo permette al progettista di un sistema di organizzare la propria mappa di memoria in modo da sfruttare la maggiore efficienza dell'indirizzamento assoluto corto per indirizzare determinate aree di memoria o alcuni

dispositivi periferici. Un metodo valido potrebbe essere quello di mettere la memoria ad accesso casuale (RAM) a partire dall'indirizzo 0 e le periferiche nei 64Kbyte più alti.

Indirizzamento assoluto lungo

Un altro tipo di indirizzamento assoluto è quello assoluto lungo, del tutto simile a quello precedente tranne per il fatto di richiedere due word di estensione per indicare il dato. È evidente che, per ridurre la lunghezza di un programma, dovrete cercare di tenere le variabili utilizzate più spesso in quelle parti della memoria che consentono l'indirizzamento assoluto corto.

Gran parte dei programmi di questo volume usano l'indirizzamento assoluto corto. Provate a modificare il valore di DATA in uno posto fuori dagli intervalli previsti dall'indirizzamento assoluto corto (ad es. 9000_{16}). Quali differenze notate nel codice oggetto? Per essere certi che l'assemblatore generi la forma assoluta corta ogni volta che è possibile, tutti i dati devono essere definiti prima del loro impiego. Provate a spostare le due pseudo-istruzioni ORG DATA e VALUE DS.W 1 alla fine del programma. Osservate il nuovo codice oggetto.

L'altro modo di indirizzamento usato in tutte le istruzioni del Programma 4-2 è quello diretto a registro dati. Ad essere interessato direttamente è, in questo caso, il contenuto del registro dati, che viene caricato, modificato o salvato, a seconda di quanto specificato nell'istruzione.

Indirizzamenti consentiti dall'istruzione MOVE

L'istruzione MOVE consente di adottare uno qualsiasi dei 14 modi di indirizzamento del processore per indicare l'operando sorgente. L'operando destinazione deve, tuttavia, essere specificato mediante quegli indirizzamenti che rimandano a locazioni di memoria "modificabili". Non sono, quindi, consentiti il modo immediato o quello relativo al contatore di programma, in quanto queste locazioni possono trovarsi nella memoria di sola lettura (ROM).

L'istruzione MOVEA

Volendo eseguire un'istruzione di tipo MOVE con un registro indirizzi come operando destinazione si deve usare l'istruzione MOVEA, che svolge la stessa funzione dell'istruzione MOVE ma senza modificare il registro di stato. Gli assemblatori della Motorola per l'MC68000 permettono di specificare un registro indirizzi come operando destinazione di un'istruzione MOVE; è l'assemblatore che provvede a generare il codice macchina relativo ad un'istruzione MOVEA, lasciando, così, immutati i flag di stato.

Il Programma 4-2 offre un altro esempio di come sia possibile sostituire due o più istruzioni con una soltanto. Le tre istruzioni di questo programma possono essere sostituite dall'unica istruzione

```
NOT.W VALUE
```

In questo caso, il contenuto della variabile VALUE viene complementato senza l'impiego di registri dati o indirizzi. Il tutto avviene direttamente sulla locazione di memoria indicata (VALUE).

4-3. Addizione a 16 bit

Scopo: Sommare il contenuto della variabile a 16 bit VALUE1, posta alla locazione 6000, al contenuto della variabile a 16 bit VALUE2, alla locazione 6002, e mettere il risultato nella variabile a 16 bit RESULT, alla locazione 6004.

Problema Campione:

```
Input:    VALUE1 - (6000) = 10F5
          VALUE2 - (6002) = 2621
Output:     RESULT - (6004) = 3716
```

Programma 4-3a:

```
00006000:                DATA     EQU     $6000
00004000:                PROGRAM  EQU     $4000
                         ;
                                  ORG     DATA
00006000:                VALUE1   DS.W    1            PRIMO VALORE
00006002:                VALUE2   DS.W    1            SECONDO VALORE
00006004:                RESULT   DS.W    1            16 BIT PER CONTENERE IL RISULTATO
                         ;
                                  ORG     PROGRAM
                         ;
00004000: 3038 6000      PGM_4_3A MOVE.W  VALUE1,D0    PRENDI IL PRIMO VALORE
00004004: D078 6002               ADD.W   VALUE2,D0    SOMMA IL SECONDO VALORE AL PRIMO
00004008: 31C0 6004               MOVE.W  D0,RESULT    SALVA IL RISULTATO
                         ;
0000400C: 4E75                    RTS
                                  END     PGM_4_3AA
```

L'istruzione ADD di questo programma è un altro esempio di istruzione con due operandi. Tuttavia, a differenza dell'istruzione MOVE, in questo caso il secondo operando non rappresenta soltanto la destinazione, ma serve anche per calcolare il risultato. Il formato

SORGENTE Operazione DESTINAZIONE → DESTINAZIONE

è comune a molte istruzioni dell'MC68000.

Uso dell'istruzione ADD

Dobbiamo ricordare a questo punto che il processore MC68000 è fornito di un bus dati esterno a 16 bit, per accedere ai dati della memoria; internamente, tuttavia, consente anche operazioni su dati a 8 ed a 32 bit. L'istruzione ADD, dunque, al pari di quella MOVE e di moltre altre, consente di operare su dati di tutte e tre le grandezze. Cambiando semplicemente il suffisso .W in .B o .L, ogni volta che compare nel programma, otterremo un programma di addizione a 8 o 32 bit.

Come abbiamo osservato nel Programma 4-1, molte istruzioni possono avere entrambi gli operandi in memoria. Questo, tuttavia, non è sempre possibile; ad esempio l'istruzione ADD permette di prelevare dalla memoria solo l'operando sorgente o quello destina-

zione. Non si può, quindi, sommare direttamente il contenuto di una locazione di memoria a quello di un'altra locazione di memoria.

Descrizione del programma 4.3b

Come avviene per ogni altro microprocessore, anche con l'MC68000 uno stesso problema può essere risolto con delle sequenze di istruzioni differenti. Il Programma 4-3b, ad esempio, è una variante del programma precedente, ed utilizza l'indirizzamento indiretto a registro indirizzi al posto di quello assoluto corto. In questo modo, non è necessario conoscere l'indirizzo dell'operando effettivo, fino al momento dell'esecuzione.

Programma 4-3b:

```
00006000:                 DATA      EQU      $6000
00004000:                 PROGRAM   EQU      $4000
                          ;
                                    ORG      DATA
00006000:                 VALUE1    DS.W     1                PRIMO VALORE
00006002:                 VALUE2    DS.W     1                SECONDO VALORE
00006004:                 RESULT    DS.W     1                16 BIT PER CONTENERE IL RISULTATO
                          ;
                                    ORG      PROGRAM

                          ;
00004000: 207C 0000
00004004: 6000            PGM_4_3B MOVEA.L  #VALUE1,A0        INIZIALIZZA A0 CON L'IND. DEL VALORE
00004006: 3010                     MOVE.W   (A0),D0           METTI IL PRIMO VALORE IN D0
00004008: D1FC 0000
0000400C: 0002                     ADDA.L   #2,A0             INCREMENTA DI 2 IL REG. INDIRIZZI A0
0000400E: D050                     ADD.W    (A0),D0           SOMMA IL SECONDO VALORE AL PRIMO
00004010: D1FC 0000
00004014: 0002                     ADDA.L   #2,A0             INCREMENTA ANCORA A0 DI 2
00004016: 3080                     MOVE.W   D0,(A0)           SALVA IL RISULTATO DELL'ADDIZIONE
                          ;
00004018: 4E75                     RTS

                                   END      PGM_4_3B
```

Indirizzamento immediato

L'istruzione MOVEA ci fa conoscere due modi di indirizzamento che non avevamo ancora impiegato: quello immediato e quello diretto a registro indirizzi. L'indirizzamento immediato vi permette di definire una costante ed includerla nel codice oggetto dell'istruzione. Il formato dell'assemblatore Motorola identifica l'indirizzamento immediato facendo precedere la costante dal segno di pound (#). La grandezza del dato varia a seconda dell'istruzione. L'indirizzamento immediato si rivela particolarmente utile con costanti di piccolo valore.

Indirizzamento diretto a registro indirizzi

L'indirizzamento diretto a registro indirizzi è analogo a quello diretto a registro dati, tranne per il fatto di utilizzare un registro indirizzi, anzichè un registro dati. Sono consentiti solo dati della grandezza di una word o di una long word. Gli operandi di una word sono sempre trasformati in valori a 32 bit mediante l'estensione del segno.

Indirizzamento indiretto a registro indirizzi

Il Programma 4-3b mostra anche l'uso dell'indirizzamento indiretto a registro indirizzi. In questo caso l'indirizzo dell'operando è contenuto nel registro indirizzi a 32 bit indicato. Dal momento che non è necessaria una word di estensione si risparmia memoria rispetto all'indirizzamento assoluto. Però dato che prima è necessario inizializzare il registro indirizzi, perchè si possa veramente parlare di risparmio di memoria il programma dovrà utilizzare questo dato diverse volte. Un altro vantaggio è rappresentato dalla mag-

giore rapidità di esecuzione rispetto all'indirizzamento assoluto. Questo è dovuto al fatto che, per poter disporre di un dato non bisogna prelevare dalla memoria la (o le) word di estensione.

Un ultimo vantaggio consiste nella flessibilità garantita dall'avere l'indirizzo in un registro indirizzi, anzichè come parte non modificabile di un'istruzione. Questo permette di utilizzare lo stesso codice con più indirizzi diversi. Perciò, volendo sommare i valori presenti in due variabili consecutive, VALUE3 e VALUE4, è sufficiente cambiare soltanto il contenuto di A0.

4-4. Shift di un bit verso sinistra

Scopo: Spostare a sinistra di un bit il contenuto della variabile a 16 bit VALUE, posta alla locazione 6000. Mettere il risultato in VALUE.

Problema Campione:

Input: VALUE – (6000) =
Output: VALUE – (6000) =

Programma 4-4:

```
00006000:                 DATA     EQU      $6000
00004000:                 PROGRAM  EQU      $4000
                          ;
                                   ORG      DATA
00006000:                 VALUE    DS.W     1              DATO DA SHIFTARE A SINISTRA
                          ;
                                   ORG      PROGRAM

                          ;
00004000: 3038 6000       PGM_4_4  MOVE.W   VALUE,D0       PRENDI IL DATO DA SHIFTARE
00004004: E348                     LSL.W    #1,D0          SHIFT LOGICO A SIN. DI UN BIT
00004006: 31C0 6000                MOVE.W   D0,VALUE       SALVA IL RISULTATO
                          ;
0000400A: 4E75                     RTS

                                   END      PGM_4_4
```

L'istruzione LSL

L'istruzione LSL esegue uno shift logico a sinistra. Con il formato dell'operando mostrato nel Programma 4-4, un registro dati può essere shiftato da 1 a 8 posizioni, utilizzando tutti i 32 bit in esso contenuti oppure solo la word o il byte di ordine basso. Un'altra versione dell'istruzione LSL consente di specificare un contatore di shift (modulo 64) in un altro registro dati. Un'ultima versione, che usa un solo operando, permette di shiftare di un bit il contenuto di una locazione di memoria, senza far uso di un registro dati.

Tranne che per i diversi effetti sul registro di stato, queste sequenze producono in D0 lo stesso risultato dell'istruzione LSL #1,D0:

```
MOVE    #1,D1
LSL     D1,D0
LSL     VALUE
MOVE    VALUE,D0
ROL     #1,D0
BCLR    #0,D0
ADD     D0,D0
```

Sareste capaci di trovarne delle altre? Quali fra quelle che vi abbiamo presentato richiedono, secondo voi, un tempo di esecuzione più breve?

4-5. Frazionamento di un byte

Scopo: Dividere il byte meno significativo della variabile ad 8 bit VALUE, posta alla locazione 6000, in due nibble da 4 bit e salvare un nibble in ciascuno dei byte della variabile a 16 bit RESULT, alla locazione 6002. I quattro bit di ordine basso del byte iniziale saranno salvati nei quattro bit di ordine basso del byte meno significativo di RESULT. I quattro bit di ordine alto sono salvati nei quattro bit di ordine basso del byte più significativo di RESULT.

Problema Campione:

Input: VALUE – (6000) = 5F
Output: RESULT – (6002) = 050F

Programma 4-5a:

```
00006000:                  DATA      EQU     $6000
00004000:                  PROGRAM   EQU     $4000
                           ;
                                     ORG     DATA
0000000F:                  MASK      EQU     $000F         MASCHERA PER IL NIBBLE DI ORD. BASSO
00006000:                  VALUE     DS.B    1             BYTE DA DISASSEMBLARE
00006001:                            DS.B    1             ALLINEA RISULTATO CON INIZIO WORD
00006002:                  RESULT    DS.W    1             LOC. PER SALVARE BYTE DISASSEMBLATO
                           ;
                                     ORG     PROGRAM
                           ;
00004000: 1038 6000        PGM_4_5A  MOVE.B  VALUE,D0      PRENDI IL BYTE DA DISASSEMBLARE
00004004: 0200 000F                  AND.B   #MASK,D0      ISOLA IL NIBBLE PIU' BASSO DEL BYTE
00004008: 11C0 6003                  MOVE.B  D0,RESULT+1   SALVA IL NIBBLE DI ORDINE BASSO
0000400C: 1038 6000                  MOVE.B  VALUE,D0      PRENDI IL BYTE DA DISASSEMBLARE
00004010: E808                       LSR.B   #4,D0         ISOLA IL NIBBLE ALTO
00004012: 11C0 6002                  MOVE.B  D0,RESULT     SALVA IL NIBBLE DI ORDINE ALTO
                           ;
00004016: 4E75                       RTS
                                     END     PGM_4_5A
```

Questo è un esempio di manipolazione di un byte. L'MC68000 permette a molte istruzioni, che operano su delle word, di operare anche su dei byte. Usando, quindi, il suffisso .B, tutte le istruzioni del Programma 4-5a agiscono su operandi da un byte.

Non dimenticate che l'istruzione MOVE, oltre a consentire trasferimenti da un registro alla memoria e viceversa, effettua trasferimenti da registro a registro. Questo tipo d'impiego è molto frequente.

Spiegazione del programma 4.5b

Generalmente si ottengono i migliori risultati, in termini di occupazione di memoria e di tempi di esecuzione, limitando i riferimenti alla memoria. Lo possiamo vedere nel Programma 4-5b, che è una variante del programma precedente.

Programma 4-5b:

```
00006000:                 DATA      EQU      $6000
00004000:                 PROGRAM   EQU      $4000
                          ;
                                    ORG      DATA

00006000:                 VALUE     DS.B     1            BYTE DA DISASSEMBLARE
00006001:                           DS.B     1            ALLINEA RISULTATO CON INIZIO WORD
00006002:                 RESULT    DS.W     1            LOC. PER SALVARE BYTE DISASSEMBLATO
                          ;
                                    ORG      PROGRAM
                          ;
00004000: 4240            PGM_4_5B  CLR.W    D0           AZZERA IL REG. DATI D0 (0:15)
00004002: 1038 6000                 MOVE.B   VALUE,D0     BYTE DA DISASSEMBLARE IN D0 (0:7)
00004006: E958                      ROL.W    #4,D0        SPOSTA IL BYTE IN D0 (4:11)
00004008: E808                      LSR.B    #4,D0        SHIFT D0 (4:7) A D0 (4:11)
0000400A: 31C0 6002                 MOVE.W   D0,RESULT    SALVA IL BYTE DISASSEMBLATO
                          ;
0000400E: 4E75                      RTS

                                    END      PGM_4_5B
```

L'istruzione CLR.W è impiegata per azzerare i 16 bit meno significativi del registro dati D0. Il trasferimento modifica soltanto il byte meno significativo di D0. L'istruzione ROL ruota la word meno significativa di D0, in modo che il nibble di ordine alto di VALUE venga a trovarsi nel secondo byte di D0. Si potrebbe usare l'istruzione ROXL al posto dell'istruzione ROL?

Convenzione sugli indirizzi di memoria

Sebbene l'MC68000 permetta di manipolare dati di dimensioni diverse, bisogna fare attenzione quando si definiscono i dati di un programma. Tutte le istruzioni del processore, quando fanno riferimento a dati di 16 0 32 bit contenuti in memoria, presumono che il bit meno significativo dell'indirizzo di memoria sia zero, che si tratti, cioè, di un'indirizzo pari. Per questo motivo, è necessario riservare un byte di memoria in più, in modo che la variabile RESULT venga a trovarsi ad un indirizzo pari (6002_{16}), anzichè alla successiva locazione di memoria disponibile, che sarebbe 6001_{16}. I risultati del Programma 4-5a sarebbero stati gli stessi, se l'indirizzo di memoria associato con RESULT fosse stato stato 6001_{16}? E cosa accadrebbe nel Programma 4-5b?

4-6. Trovare il maggiore di due numeri

Scopo: Trovare la maggiore di due variabili a 32 bit, VALUE1 (alla locazione 6000) e VALUE2 (alla locazione 6004). Mettere il risultato nella variabile RESULT, alla locazione 6008. Si presuppone che si tratti di valori privi di segno.

Problema Campione:

a. Input:	VALUE1	–	(6000)	=	12345678
	VALUE2	–	(6004)	=	87654321
Output:	RESULT	–	(6008)	=	87654321

b. Input:	VALUE1	–	(6000)	=	12345678
	VALUE2	–	(6004)	=	0ABCDEF1
Output:	RESULT	–	(6008)	=	87654321

Programma 4-6:

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
                                     ORG      DATA
00006000:                  VALUE1    DS.L     1              PRIMO VALORE
00006004:                  VALUE2    DS.L     1              SECONDO VALORE
00006008:                  RESULT    DS.L     1              RISERVA LONG WORD PER IL RISULTATO
                           ;
                                     ORG      PROGRAM

                           ;
00004000: 4CF8 0003        PGM_4_6   MOVEM.L  VALUE1,D0/D1   CARICA I VALORI DA CONFRONTARE
00004004: 6000                       CMP.L    D0,D1          CONFRONTA I DUE VALORI A 32 BIT
00004006: B280                       BHI      FINI           SE VALUE2 >= VALUE1 VAI A FINI
00004008: 6200 0004                  MOVE.L   D0,D1          ...ALTRIMENTI D1= VALUE1
0000400C: 2200             FINI      MOVE.L   D1,RESULT      SALVA IL VALORE PIU' GRANDE
0000400E: 21C1 6008        ;
00004012: 4E75                       RTS

                                     END      PGM_4_6
```

L'istruzione MOVEM

L'istruzione MOVE Multipla, MOVEM, usata nel Programma 4-6 ci permette di trasferire il contenuto di un determinato registro indirizzi/dati a o da un blocco di locazioni di memoria consecutive. Nel Programma 4-6 in D0 e D1 viene caricato, mediante l'istruzione MOVEM, il contenuto, rispettivamente, delle variabili VALUE1 e VALUE2.

Sebbene sia possibile specificare quali registri utilizzare con un'istruzione MOVEM, l'ordine in cui vengono trasferiti i contenuti dei registri non è definibile del programmatore. L'ordine di trasferimento inizia dal registro dati D0 (o dal registro dati con il numero più basso fra quelli impiegati), prosegue fino al registro D7 (o fino al registro dati con il numero più alto) e continua con i registri indirizzi da A0 ad A7 (o comunque con quelli che vengono utilizzati). La sola eccezione si verifica quando viene adottato l'indirizzamento con

predecremento; in tal caso, l'ordine è completamente opposto a quello appena descritto. Per maggiori dettagli sul modo in cui indicare i registri, vi rimandiamo alla descrizione dell'istruzione MOVE, nel Capitolo 22.

L'istruzione CMP

L'istruzione Compare, CMP, del Programma 4-6 fa sì che i flag del registro di stato assumano gli stessi valori che avrebbero se l'operando sorgente, D0, fosse sottratto dal registro destinazione, D1. L'ordine degli operandi è identico a quello di un'istruzione di sottrazione, SUB.

L'istruzione BHI

L'istruzione di trasferimento condizionale BHI trasferisce il controllo all'istruzione contraddistinta dalla label FINI, se il contenuto privo di segno di D1 è maggiore o uguale al contenuto di D0. In caso contrario, viene eseguita l'istruzione successiva (MOVE.L D0,D1). In ogni caso, in corrispondenza dell'istruzione con la label FINI, il registro D1 conterrà sempre il maggiore dei due valori.

L'istruzione BHI è una delle quattordici istruzioni di salto condizionato Per modificare il programma in modo che operi su numeri provvisti di segno, basta semplicemente sostituire BHI con BGE:

```
—
—
CMP.L   D0,D1
BGE     FINI
—
—
```

La tabella seguente indica quali sono le istruzioni di salto condizionato da utilizzare a seconda che vengano confrontati valori con o senza segno:

Condizione	Con segno	Senza segno
maggiore o uguale a	BGT	BHI
maggiore di	BGE	BCC
uguale a	BEQ	BEQ
non uguale a	BNE	BNE
minore o uguale a	BLS	BLS
minore di	BLT	BCS

Sono utilizzate le stesse istruzioni per l'addizione, la sottrazione o il confronto di valori provvisti o privi di segno; tuttavia, le modalità del confronto sono diverse.

Istruzione di salto condizionato

Le istruzioni di salto condizionato sono un esempio di indirizzamento relativo al contatore di programma. In altre parole, se una determinata condizione è soddisfatta il controllo viene trasferito ad un indirizzo relativo al valore attuale del contatore di programma. L'MC68000 consente di utilizzare, come valori di spostamento, grandezze di 8 o di 16 bit. Dal momento che lo spostamento è dato da un valore espresso in complemento a due ed avviene solo dopo che il contatore di programma è stato incrementato, con le istruzio-

ni di salto è possibile spostarsi indietro fino ad un massimo di126 o 32766 byte ed in avanti fino a 128 o 32768 byte.

Le istruzioni di confronto e diramazione sono una componente importante nella programmazione dell'MC68000. Non bisogna confondere il significato di un'istruzione CMP. **Effettuato un confronto, la relazione valutata è:**

DESTINAZIONE condizione SORGENTE

Ad esempio, se la condizione è "minore di", è necessario controllare che l'operando destinazione sia minore dell'operando sorgente. Cercate di acquistare una certa familiarità con le varie condizioni ed i loro significati. I confronti fra valori privi di segno sono molto utili quando è necessario confrontare fra loro due indirizzi.

4-7. Addizione a 64 bit

Scopo: Sommare il contenuto di due variabili a 64 bit, VALUE1 (alla locazione 6000) e VALUE2 (alla locazione 6008). Mettere il risultato in RESULT (alla locazione 6010).

Problema Campione:

Input:	VALUE1 -	(6000)	= 12A2	
		(6002)	= E640	12A2EE640F210123
		(6004)	= F210	
		(6006)	= 0123	
	VALUE2 -	(6008)	= 0010	001019BF40023F51
		(600A)	= 19BF	
		(600C)	= 4002	
		(600E)	= 3F51	
Output:	RESULT -	(6010)	= 12B3	
		(6012)	= 0000	2B3000032124074
		(6014)	= 3212	
		(6016)	= 4074	

Programma 4-7:

```
00006000:                   DATA     EQU      $6000
00004000:                   PROGRAM  EQU      $4000
                            ;
                                     ORG      DATA
00006000:                   VALUE1   DS.L     2                 PRIMO VALORE
00006008:                   VALUE2   DS.L     2                 SECONDO VALORE
00006010:                   RESULT   DS.L     2                 RISERVA 64 BIT PER IL RISULTATO
                            ;
                                     ORG      PROGRAM

                            ;
00004000: 4CF8 000F
00004004: 6000              PGM_4_7  MOVEM.L VALUE1,D0-D3       D0-D1:= VALUE1 E D2-D3:= VALUE2
00004006: D283                       ADD.L    D3,D1             SOMMA LE LONG WORD MENO SIGNIFICATIVE
00004008: D182                       ADDX.L   D2,D0             SOMMA L.W. PIU' SIGNIF.CON ESTENSIONE
0000400A: 48F8 0003
0000400E: 6010                       MOVEM.L D0-D1,RESULT       SALVA I 64 BIT DEL RISULTATO
                            ;
00004010: 4E75                       RTS

                                     END      PGM_4_7
```

Spiegazione del programma 4.7

L'utilità dell'istruzione Move Multipla (MOVEM) risulta evidente anche in questo trasferimento a 128 bit nei registri dati D0-D3. I flag del registro di stato restano immutati, in seguito a questo trasferimento; mentre l'istruzione ADD modifica sia il Carry che l'Extend. La condizione del flag di Extend è utilizzata nell'istruzione ADDX (Add with Extend) per includere nella somma il riporto della precedente addizione a 32 bit.

4-8. Tabella di fattoriali

Scopo: Calcolare il fattoriale della variabile ad 8 bit VALUE, alla locazione 6010, mediante una tabella di fattoriali FTABLE, che occupa le locazioni di memoria da 6000 a 600F. Mettere il risultato nella variabile a 16 bit RESULT, alla locazione 6021. Si presume la presenza in VALUE di un valore compreso fra 0 e 7.

Problema Campione:

Input:	FTABLE-	(6000)	= 0000 0!	= 1_{10}
		(6002)	= 0001 1!	= 1_{10}
		(6004)	= 0002 2!	= 2_{10}
		(6006)	= 0006 3!	= 6_{10}
		(6008)	= 0018 4!	= 24_{10}
		(600A)	= 0078 5!	= 120_{10}
		(600C)	= 02D0 6!	= 720_{10}
		(600E)	= 13B0 7!	= 5040_{10}
	VALUE-	(6010)	= 05	
Output:	RESULT-	(6012)	= 0078 5!	= 120_{10}

Programma 4-8a:

```
00006000:                  DATA     EQU      $6000
00004000:                  PROGRAM  EQU      $4000
                           ;
                                    ORG      DATA

                           *    TABELLA DI FATTORIALI
                           ;
00006000:                  FTABLE   DC       1                0! := 1
00006002:                           DC       1                1! := 1
00006004:                           DC       2                2! := 2
00006006:                           DC       6                3! := 6
00006008:                           DC       24               4! := 24
0000600A:                           DC       120              5! := 120
0000600C:                           DC       720              6! := 720
0000600E:                           DC       5040             7! := 5040
                           ;
00006010:                  VALUE    DS.B     1                TROVA IL FATTORIALE DI QUESTO VAL.
00006011:                           DS.B     1                ALLINEAMENTO
00006012:                  RESULT   DS.W     1                RISULTATO DEL FATTORIALE
                           ;
                                    ORG      PROGRAM

00004000: 4240             PGM_4_8A CLR.W    D0               D0 (0:15) := 0
00004002: 1038 6010                 MOVE.B   VALUE,D0         PRENDI IL DATO
00004006: D000                      ADD.B    D0,D0            D0 (0:7) := 2 * VALUE
00004008: 307C 6000                 MOVEA.W  #FTABLE,A0       INIZ. PUNTATORE TAB. FATTORIALI
0000400C: 31F0 0000
00004010: 6012                      MOVE.W   0(A0,D0),RESULT  SALVA IL RISULTATO DEL FATTORIALE
                           ;
00004012: 4E75                      RTS

                                    END      PGM_4_8A
```

Spiegazione del programma 4.8a

Il metodo impiegato per la consultazione di una tabella nel Programma 4-8a mostra l'uso dell'indirizzamento indiretto a registro indirizzi con indice. Le prime due istruzioni, CLR e MOVE, caricano il registro indice con il contenuto di VALUE. È richiesta l'istruzione CLR, perchè la grandezza del dato VALUE è pari ad un byte, mentre la grandezza del registro indice usato in questo tipo di indirizzamento è di una word o di una long word. L'MC68000 permette di utilizzare, come indice, sia un registro dati che un registro indirizzi.

L'istruzione Move Address (MOVEA) inizializza il registro indirizzi A0 con l'indirizzo della tabella di fattoriali. Sono interessati tutti i 32 bit del registro indirizzi, indipendentemente dalle dimensioni del dato. Quando si tratta di una word, come nel Programma 4-8a, l'operando sorgente viene trasformato in un valore a 32 bit, mediante l'estensione del segno.

La posizione esatta all'interno della tabella dell'elemento desiderato è determinata dal primo operando dell'istruzione MOVE.W. La long word contenuta nel registro indirizzi A0 viene sommata alla word del registro dati D0, dopo averne esteso il segno, per ottenere l'indirizzo effettivo. In questo caso, è D0 ad essere impiegato come registro indice. Come avviene nella quasi totalità dei modi di indirizzamento, l'uso di un registro dati o indirizzi, allo scopo di definire un indirizzo effettivo, non ne altera il contenuto. Gli indirizzamenti diretti, con postincremento e con predecremento rappresentano un'eccezione a questa regola.

Il modo indiretto a registro indirizzi con indice permette di utilizzare, nel calcolo dell'indirizzo, 16 o 32 bit del registro indice. La grandezza del registro indice è specificata dal suffisso dell'operando, che indica il registro indice. Come per le istruzioni, il suffisso

di default è '.W' o word. Perchè non può essere usato il suffisso .L per il registro indice D0 nel Programma 4-8a?

Oltre a permettere di stabilire l'indirizzo effettivo mediante il contenuto del registro indirizzi e del registro indice, il modo indiretto a registro indirizzi con indice consente anche un piccolo spostamento. Il campo di spostamento prevede un valore di 8 bit, che, essendo provvisto di segno, rende possibili spostamenti compresi fra −126 e +129 byte.

Programma 4-8b:

```
00006000:                  DATA     EQU      $6000
00004000:                  PROGRAM  EQU      $4000
                           ;
                                    ORG      DATA

                           *    TABELLA DI FATTORIALI
                           ;
00006000:                  FTABLE   DC       1              0!  := 1
00006002:                           DC       1              1!  := 1
00006004:                           DC       2              2!  := 2
00006006:                           DC       6              3!  := 6
00006008:                           DC       24             4!  := 24
0000600A:                           DC       120            5!  := 120
0000600C:                           DC       720            6!  := 720
0000600E:                           DC       5040           7!  := 5040
                           ;
00006010:                  VALUE    DS.B     1              TROVA IL FATTORIALE DI QUESTO VAL.
00006011:                           DS.B     1              ALLINEAMENTO
00006012:                  RESULT   DS.W     1              RISULTATO DEL FATTORIALE
                           ;
                                    ORG      PROGRAM

00004000: 4240             PGM_4_8B CLR.W    D0             D0 (0:15) := 0
00004002: 1038 6010                 MOVE.B   VALUE,D0       PRENDI IL DATO
00004006: D000                      ADD.B    D0,D0          D0 (0:7) := 2 * VALUE
00004008: 3040                      MOVEA.W  D0,A0          SPOSTA L'OFFSET NEL REG. IND. A0
0000400A: 31E8 6000
0000400E: 6012                      MOVE.W   FTABLE(A0),RESULT SALVA IL RISULTATO
                           ;
00004010: 4E75                      RTS

                                    END      PGM_4_8B
```

Spiegazione del programma 4.8b

Il Programma 4-8b è del tutto identico al Programma 4-8a, tranne che per il fatto di utilizzare un altro modo di indirizzamento, quello indiretto a registro indirizzi con spostamento. In questo caso, l'indirizzo effettivo dell'operando è la somma del registro indirizzi e del valore di spostamento a 16 bit, con estensione del segno, contenuto nella word di estensione, che segue l'istruzione nella memoria di programma.

Nel Programma 4-8b lo "spostamento" è, in realtà, la base della tabella, mentre il registro indirizzi viene usato per spostarsi all'interno della tabella stessa. È importante ricordare che il valore di spostamento è un valore a 16 bit che, prima di essere utilizzato, subisce l'estensione del segno. Perciò, se FTABLE si fosse trovata ad un qualsiasi indirizzo maggiore di $7FFF_{16}$, l'estensione del segno avrebbe fatto sì che l'inizio della tabella diventasse un indirizzo compreso fra $FF8000_{16}$ e $FFFFFF_{16}$. Così, ad esempio, il Programma 4-8b non funzionerebbe se FTABLE si trovasse all'indirizzo 015000_{16}. Il metodo di usare lo "spostamento" come indirizzo di base si rivela utile solo nel caso che questo indirizzo si trovi tra 0 e $7FFF_{16}$ oppure tra $FF8000_{16}$ e $FFFFFF_{16}$.

Il modo in cui viene impiegato l'indirizzamento indiretto a registro indirizzi con spostamento nel Programma 4-8b non è quello tipico di questo tipo di indirizzamento. Normalmente, il registro indirizzi contiene l'indirizzo di una tabella o di una serie di dati strutturati e si è soliti servirsi di un valore di spostamento costante rispetto all'inizio della tabella o della struttura.

PROBLEMI

4-1. Trasferimento di un dato a 64 bit

Scopo: Trasferire il contenuto delle locazioni di memoria da 6000 a 6006 alle locazioni 6800-6806.

Problema Campione:

Input:	(6000) = 3E2A
	(6002) = 42A1
	(6004) = 21F2
	(6006) = 60A0
Output:	(6800) = 3E2A
	(6802) = 42A1
	(6804) = 21F2
	(6806) = 60A0

4-2. Sottrazione a 16 bit

Scopo: Sottrarre il contenuto della variabile a 16 bit VALUE1, alla locazione 6000, dal contenuto della variabile a 16 bit VALUE2, alla locazione 6002, e salvare il risultato in VALUE1.

Problema Campione:

Input:	VALUE1 – (6000) = 3977
	VALUE2 – (6002) = 2182
Output:	VALUE1 – (6000) = 17F5

4-3. Shift di tre bit verso destra

Scopo: Shiftare a destra di tre bit il contenuto della variabile a 16 bit VALUE1, alla locazione 6000. Azzerare i tre bit più significativi.

Problema Campione:

a.	Input:	VALUE1	– (6000)	= 415D
	Output:	VALUE1	– (6000)	= 082B
b.	Input:	VALUE1	– (6000)	= C15D
	Output:	VALUE1	– (6000)	= 182B

4-4. Assemblaggio di una word

Scopo: Combinare i quattro bit di ordine basso di ciascuno dei quattro byte consecutivi, a cominciare dalla locazione 6000, in un'unica word da 16 bit. Il valore di 6000 occuperà il nibble più significativo del risultato; il valore di 6003 diventerà il nibble meno significativo. Salva il risultato nella locazione 6004.

Problema Campione:

Input:	(6000) = 0C
	(6001) = 02
	(6002) = 06
	(6003) = 09
Output:	(6004) = C269

4-5. Trovare il più piccolo fra tre numeri

Scopo: Le locazioni 6000, 6002 e 6004 contengono ciascuna un numero privo di segno. Mettere il più piccolo di questi numeri nella locazione 6006.

Problema Campione:

Input:	(6000) = 9125
	(6002) = 102C
	(6004) = 7040
Output:	(6006) = 102C

4-6. Somma di quadrati

Scopo: Calcolare i quadrati del contenuto della word VALUE1, alla locazione 6000, e della word VALUE2, alla locazione 6002, e sommarli tra loro. Mettere il risultato nella long word RESULT, alla locazione 6004. Si presuppone che si tratti di numeri forniti di segno.

Problema Campione:

```
Input:  VALUE1  - (6000)  = 0007
        VALUE2  - (6002)  = 0032
Output: RESULT  - (6004)  = 000009F5
```

Cioè, $7 + 502 = 49 + 2500 = 2549$ (decimale)

$7^2 + 32^2 = 31 + 9C4 = 9F5$ (esadecimale)

Soluzione:

```
MOVE.W    VALUE1,D0
MULS.W    VALUE1,D0
MOVE.W    VALUE2,D1
MULS.W    VALUE2,D1
ADD.L     D0,D1
MOVE.L    D1,RESULT
```

4-7. Shift verso sinistra di un numero variabile di bit

Scopo: Eseguire lo shift a sinistra del contenuto della word VALUE, alla locazione di memoria 6000. Il numero di posizioni da shiftare è contenuto nella word COUNT, alla locazione 6002. Si deve trattare di un valore minore di 32. I bit di ordine basso devono essere azzerati.

Problema Campione:

```
a.   Input:  (6000) = 182B
             (6002) = 0003     shift a sinistra di 3 posizioni
    Output:  (6000) = C158

b.   Input:  (6000) = 182B
             (6002) = 0010     shift a sinistra di 16 posizioni
    Output:  (6000) = 0000
```

Soluzione:

```
MOVEM.W   VALUE,D0/D1
LSL.W     D1,D0
MOVE.W    D0,VALUE
```

CAPITOLO 5

CICLI DI PROGRAMMA

Definizione di "loop"

Un ciclo di programma (o "loop") è la struttura fondamentale che costringe la CPU a ripetere una sequenza di istruzioni. Un loop è formato da quattro parti:

1. **La sezione di inizializzazione,** che stabilisce i valori iniziali dei contatori, dei puntatori e delle altre variabili.
2. **La sezione di elaborazione,** dove avviene la reale manipolazione dei dati. Questa è la sezione che svolge il lavoro vero e proprio.
3. **La sezione di controllo del ciclo,** che aggiorna i contatori ed i puntatori per la successiva ripetizione del ciclo.
4. **La sezione conclusiva,** necessaria ad analizzare e salvare i risultati.

Il computer esegue solo una volta le Sezioni 1 e 4 mentre ripete più volte le Sezioni 2 e 3. Perciò, il tempo di esecuzione del loop dipende essenzialmente dalla durata delle Sezioni 2 e 3 che devono essere quanto più rapide possibile, mentre i tempi di esecuzione delle Sezioni 1 e 4 hanno una minore incidenza sulla velocità complessiva del programma.

Le Figure 5-1 e 5-2 contengono due diagrammi di flusso alternativi per un tipico ciclo di programma. In base al diagramma della Figura 5-1 il computer esegue la sezione di elaborazione almeno una volta. Mentre nel caso della Figura 5-2 la fase di elaborazione può anche non essere eseguita. La sequenza delle operazioni mostrata nella Figura 5-1 è più naturale, ma quella della Figura 5-2 garantisce spesso una maggiore efficienza, evitando che il computer esegua la sequenza di elaborazione, anche quando non è necessario.

Il computer si serve di un loop per elaborare grossi insiemi di dati (chiamati anche "array"). Il modo più semplice di usare una sequenza di istruzioni per la gestione di un array è quello di incrementare un registro (di solito un registro indice o un puntatore di stack), dopo ogni iterazione. Perciò quando il computer ripeterà la sequenza delle istruzioni quel registro conterrà l'indirizzo dell'elemento successivo. Un computer può, in questo modo, gestire array di qualsiasi lunghezza, all'interno di un unico programma.

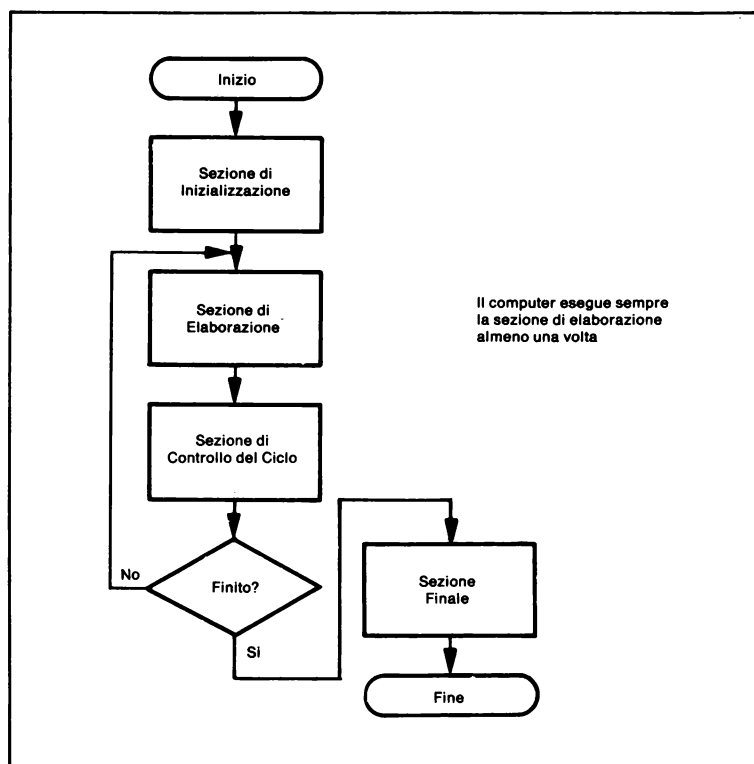



Figura 5-1. Diagramma di Flusso di un Ciclo di Programma

L'indirizzamento indiretto a registro è la chiave per il trattamento di array con il microprocessore MC68000, in quanto consente di variare l'indirizzo reale del dato (o "indirizzo effettivo"), cambiando soltanto il contenuto di un registro. Nell'indirizzamento assoluto, è l'istruzione ad indicare l'indirizzo effettivo, che non può essere modificato, quando il programma si trova nella memoria di sola lettura.

L'indirizzamento con autoincremento è particolarmente adatto per gestire degli array, dal momento che provvede automaticamente ad aggiornare il registro indirizzi per l'iterazione successiva, senza nessuna istruzione addizionale. Si possono ottenere incrementi di 2 o di 4, a seconda che l'array contenga dati o indirizzi a 16 o 32 bit.

I due tipi fondamentali di loop

Sebbene nei nostri esempi sia mostrata l'elaborazione di array con autoincremento (sommando F, 2 o 4 dopo ogni iterazione), **la procedura è valida anche nel caso dell'autodecremento** (sottraendo 1, 2 o 4 prima di ogni iterazione). Molti programmatori trovano che leggere un array alla rovescia sia poco elegante e più complesso, ma in molte occasioni si rivela un metodo particolarmente efficace. Un computer evidentemente non distingue fra indietro e avanti. **Il programmatore, però, deve ricordare che l'MC68000 incrementa un registro indirizzi, dopo averlo usato, ma lo decrementa, prima di**

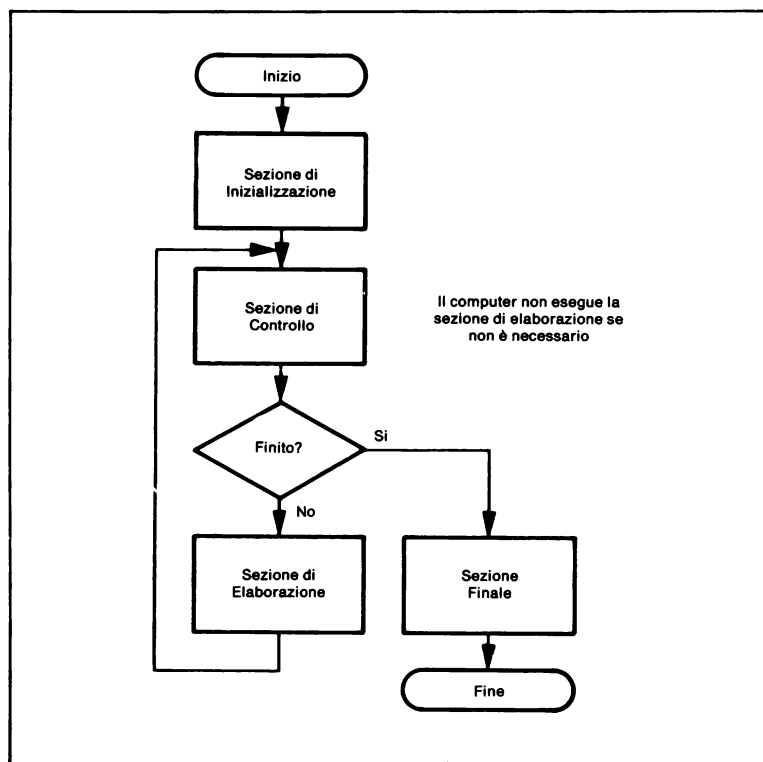



Figura 5-2. Un Ciclo di Programma Alternativo

usarlo. Questa differenza rende necessarie due fasi di inizializzazione diverse:

Inizializzazione del registro indirizzi nella gestione di un array

1. Se ci spostiamo in avanti, il registro indirizzi dovrà essere inizializzato con l'indirizzo più basso occupato dall'array.
2. Spostandoci indietro (autodecremento) bisognerà inizializzare il registro indirizzi con un indirizzo superiore (di 1, 2 o 4) rispetto a quello più alto occupato dall'array.

ESEMPI DI PROGRAMMAZIONE

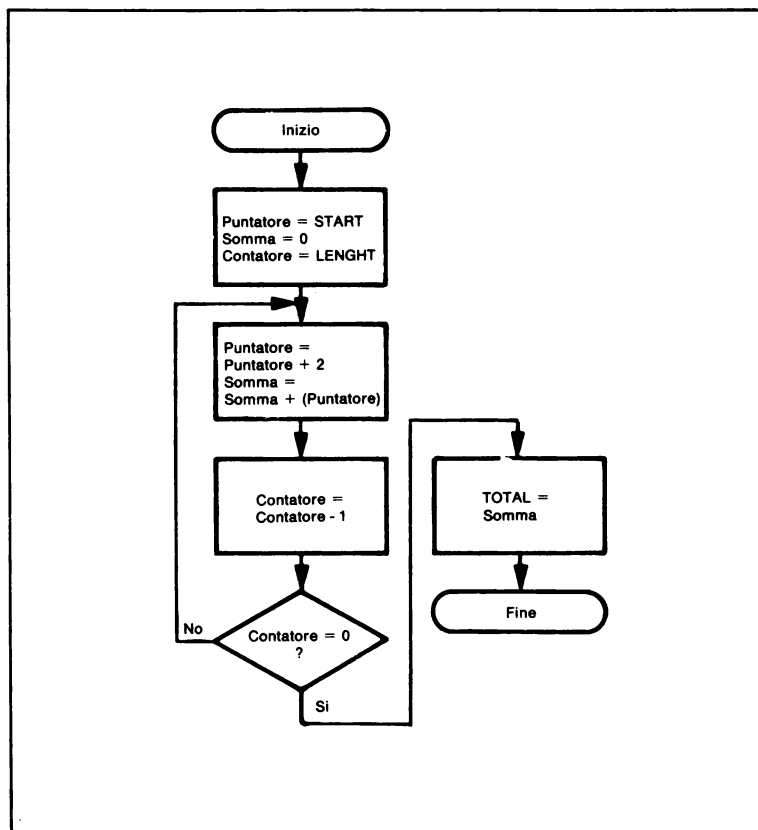
5-1. Somma a 16 bit di una serie di dati

Scopo: Calcolare la somma di una serie di numeri. La lunghezza della serie (in word) è definita dalla variabile LENGTH, alla locazione 6000. L'indirizzo iniziale della serie è contenuto nella variabile long word START, alla locazione

6002. Salvare il risultato nella variabile TOTAL, alla locazione 6006. Il risultato è un numero a 16 bit, per cui i riporti possono essere ignorati.

Problema Campione:

Input:	LENGTH	– (6000)	=	0003
	START	– (6002)	=	00005000
		(5000)	=	2040
		(5002)	=	1C22
		(5004)	=	0242
Output:	TOTAL	– (6006) (5004)	=	(5000) + (5002) +
			=	2040 + 1C22 + 0242
			=	3EA4

Diagramma di Flusso 5-1

Programma 5-1a:

```
00006000:                   DATA      EQU      $6000
00004000:                   PROGRAM   EQU      $4000
                            ;
                                      ORG      DATA
00006000:                   LENGTH    DS.W     1              NUMERO DEI DATI
00006002:                   START     DS.L     1              INDIRIZZO DI BASE DEI DATI
00006006:                   TOTAL     DS.W     1              SOMMA DEGLI ELEMENTI
                            ;
                                      ORG      PROGRAM

00004000: 2078 6002         PGM_5_1A  MOVEA.L  START,A0       INIZIALIZZA IL PUNTATORE
00004004: 7000                        MOVEQ    #0,D0          INIZIALIZZA LA SOMMA A 0
00004006: 3238 6000                   MOVE.W   LENGTH,D1      INIZIALIZZA IL CONTATORE
                            ;
0000400A: D058              LOOP      ADD.W    (A0)+,D0       AGGIUNGI IL DATO SEGUENTE
0000400C: 5341                        SUBQ.W   #1,D1          AGGIORNA IL CONTATORE
0000400E: 66FA                        BNE      LOOP           SE NON E' ZERO VAI A LOOP
                            ;
00004010: 31C0 6006                   MOVE.W   D0,TOTAL       SALVA IL RISULTATO
                            ;
00004014: 4E75                        RTS

                                      END      PGM_5_1A A
```

Spiegazione del programma 5.1a

La fase di inizializzazione è rappresentata dalle prime tre istruzioni che assegnano gli opportuni valori iniziali al puntatore ai dati, alla somma ed al contatore. In questo programma troviamo il primo esempio di un passaggio di parametri, che, in questo caso, sono un indirizzo (il contenuto di START) e una dimensione o contatore (LENGTH), che abbiamo già incontrato nei programmi precedenti. La prima istruzione MOVE carica l'indirizzo iniziale dei dati (dalla locazione START) nel registro indirizzi A0. Nei Capitoli 10 e 11 troverete maggiori dettagli sulle modalità per il passaggio dei parametri. Per il momento è sufficiente che vi accertiate, prima di eseguire il programma, che l'indirizzo iniziale richiesto si trovi nella long word alla locazione 6002.

Spesso, capita di dover inizializzare un registro dati con un piccolo valore, come abbiamo fatto nel Programma 5-1a. Per valori compresi fra -128 e + 127, ci serviremo dell'istruzione MOVEQ, che codifica il valore nella stessa word dell'istruzione, eliminando, così, la word dell'operando, che, altrimenti, sarebbe stata necessaria per definire il valore iniziale. Osservate come l'istruzione MOVEQ, a differenza della maggior parte delle istruzioni dell'MC68000, preveda soltanto dati di grandezza pari ad una long word. Potevamo utilizzare anche l'istruzione CLR per inizializzare la somma a zero; sia MOVEQ che CLR richiedono un uguale numero di byte e di cicli del microprocessore. In quali casi è preferibile usare l'istruzione CLR?

La sezione di elaborazione, nel Programma 5-1a, è costituita dalla sola istruzione ADD.W (A0)+,D0 che somma il contenuto della locazione di memoria il cui indirizzo è contenuto nel registro indirizzi A0 al contenuto del registro dati D0. È questa l'istruzione che svolge la funzione più importante ed è il primo esempio di indirizzamento indiretto a registro indirizzi con postincremento. Avrete, probabilmente, notato che il programma non contiene un'istruzione che si occupa esplicitamente di aggiornare il registro indirizzi in modo da puntare alla word successiva della serie. L'ag-

giornamento avviene implicitamente, durante l'esecuzione dell'istruzione ADD, che, proprio per questo motivo, fa parte anche della sezione di controllo. Nell'indirizzamento con postincremento il processore incrementa il registro indirizzi, dopo averlo utilizzato per determinare l'indirizzo effettivo della locazione contenente il dato. Il contenuto del registro indirizzi è incrementato di 1, 2 o 4, a seconda delle dimensioni di un dato (1 se si tratta di un byte, 2 per una word e 4 per una long word). Perciò, l'istruzione ADD.W (A0)+,D0 incrementa di 2 il contenuto del registro indirizzi A0. Questo tipo di indirizzamento è particolarmente utile quando utilizziamo delle tabelle di dati.

La sezione di controllo vera e propria consiste unicamente dell'istruzione SUBQ.W, dal momento che l'istruzione ADD.W (A0)+,D0, come abbiamo visto, aggiorna automaticamente il puntatore. L'istruzione SUBQ.W decrementa il contatore relativo al numero di iterazioni, che devono essere ancora eseguite. L'istruzione Subtract Quick (SUBQ.W) riduce la lunghezza del programma, in quanto permette, al pari dell'istruzione MOVEQ, di codificare dei piccoli valori all'interno di un'unica word d'istruzione. A differenza di MOVEQ, SUBQ consente valori compresi fra 1 e 8; può, tuttavia, operare su dati di un byte, di una word o di una long word e direttamente sulla memoria.

L'istruzione BNE causa una diramazione se il flag di Zero (Z) del registro di stato è posto a 0 (se, cioè, il risultato del decremento non è zero). Lo spostamento è indicato da un numero in complemento a due, che dipende dalla distanza fra l'istruzione e la destinazione. In questo caso, si tratta della distanza fra la locazione di memoria 4010 (l'indirizzo dell'istruzione BNE + 2) e la locazione 400A (la destinazione). Quindi, l'offset, usando un numero in complemento a due, sarà:

```
      400A          400A
 - (400E+2)    =   +BFF0
                  --------
                    FFFA
```

Il valore di spostamento $FA corrisponde ad un sei negativo (–6), che rappresenta il numero di byte fra la label LOOP e la locazione con l'istruzione di salto, più due. L'impiego di un solo byte con estensione del segno permette spostamenti compresi fra -63 e +64 word, rispetto alla locazione con l'istruzione di salto. Il valore viene espresso in word, anzichè in byte, poichè tutte le istruzioni dell'MC68000 devono coincidere con l'inizio di una word ed hanno dimensioni che sono multipli di word. Un'altra forma dell'istruzione di salto prevede un offset di 16 bit con estensione del segno e la possibilità di spostamenti compresi tra –16383 e +16384 word. In questo caso è necessaria un'ulteriore word di operando.

Se il flag di Zero è 1 (se, cioè, il risultato del decremento di D1 è

zero), il processore continua nella sua normale sequenza. Il risultato dell'esecuzione di BNE sarà dunque:

PC =
- LOOP se il risultato del decremento di D1 non è zero
- (PC) + 2 se il risultato del decremento di D1 è zero

Il 2 in più è dovuto, come al solito, ai due byte occupati dall'istruzione BNE. Questo vale per tutte e due le versioni dell'istruzione di salto, poichè, in entrambi i casi, il PC è incrementato di due prima di aggiungervi il valore di spostamento. Con un offset di 16 bit, se non si verifica il salto, il PC è incrementato ancora di due. Il risultato è lo stesso sia per un valore di spostamento di 8 che di 16 bit: se la condizione indicata non è soddisfatta, viene eseguita l'istruzione immediatamente successiva.

Sono molti i programmatori che, in un ciclo di programma, preferiscono decrementare un contatore, anzichè incrementarlo, in modo da usare il valore del flag di Zero come condizione di uscita. Come si ricorderà il flag di Zero vale 1, se l'ultimo risultato era zero, altrimenti vale 0. Provate a riscrivere il programma, caricando inizialmente il registro D1 con zero ed incrementandolo dopo ogni iterazione. Quale dei due metodi risulta più efficace?

Il Programma 5-1a funziona correttamente qualunque siano i valori iniziali, tranne che con un numero di elementi uguale a zero. Si può risolvere il problema inserendo un opportuno controllo per un'eventualità di questo tipo prima che abbia inizio il loop, come nel Programma 5-1b.

Programma 5-1b

```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
                                  ORG      DATA
00006000:                LENGTH   DS.W     1          NUMERO DEI DATI
00006002:                START    DS.L     1          INDIRIZZO DI BASE DEI DATI
00006006:                TOTAL    DS.W     1          SOMMA DEGLI ELEMENTI
                         ;
                                  ORG      PROGRAM

00004000: 2078 6002      PGM_5_1B MOVEA.L  START,A0   INIZIALIZZA IL PUNTATORE
00004004: 7000                    MOVEQ    #0,D0      INIZIALIZZA LA SOMMA A 0
00004006: 3238 6000               MOVE.W   LENGTH,D1  INIZIALIZZA IL CONTATORE
                         ;
0000400A: 6706                    BEQ.S    DONE       SE LENGTH = 0 ALLORA DONE
0000400C: D058           LOOP     ADD.W    (A0)+,D0   AGGIUNGI IL DATO SEGUENTE
0000400E: 5341                    SUBQ.W   #1,D1      AGGIORNA IL CONTATORE
00004010: 66FA                    BNE      LOOP       SE NON E' ZERO VAI A LOOP
                         ;
00004012: 31C0 6006      DONE     MOVE.W   D0,TOTAL   SALVA IL RISULTATO
                         ;
00004016: 4E75                    RTS

                                  END      PGM_5_1B
```

Spiegazione del programma 5.1b

L'istruzione BEQ verifica se il numero degli elementi è uguale a zero e, in tal caso, il controllo del programma è trasferito a DONE. Avrete notato che l'istruzione di salto BEQ ha un suffisso ".S", che è utilizzato dall'assemblatore per stabilire il tipo di offset di un'istruzione di salto. Questo suffisso è necessario solamente quando la

label, nel campo dell'operando, è definita solo successivamente e l'opzione di default dell'assembler è un offset di tipo lungo.

L'ordine in cui il processore esegue le istruzioni è spesso molto importante. Nel Programma 5-1b, BEQ segue immediatamente l'istruzione MOVE.W LENGTH,D1; altrimenti, la presenza di un'istruzione intermedia potrebbe modificare il flag di Zero. Per lo stesso motivo, l'istruzione SUBQ deve essere seguita immediatamente dall'istruzione BNE.

5-2. Somma a 32 bit di una serie di dati

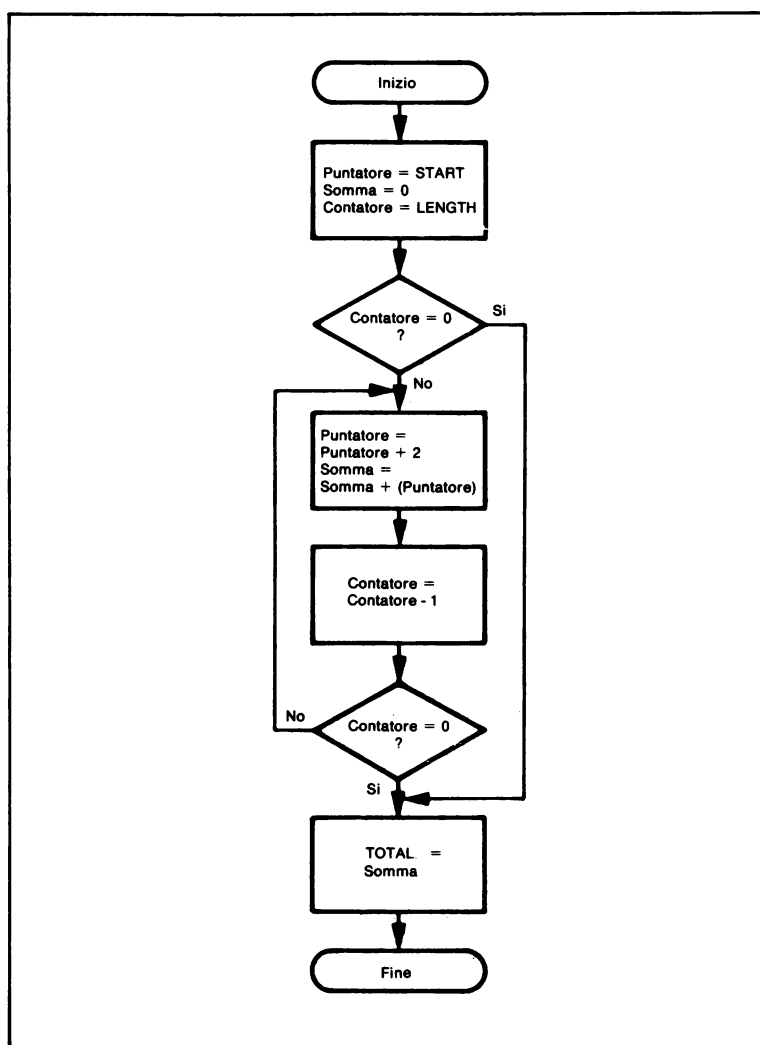
Scopo: Calcolare la somma di una serie di numeri a 16 bit, privi di segno. La lunghezza della serie (in word) è definita dalla variabile LENGTH, alla locazione 6000. L'indirizzo iniziale della serie è contenuto nella variabile long word START, alla locazione 6002. Salvare il risultato nella variabile long word TOTAL, alla locazione 6006. Tenere conto dei riporti.

Problema Campione:

```
Input:   LENGTH   - (6000) = 0003
         START    - (6002) = 00005000
                    (5000) = 2040
                    (5002) = 1C22
                    (5004) = E242
Output:  TOTAL    - (6006) = (5000) + (5002) + (5004)
                           = 2040 + 1C22 + E242
                           = 00011EA4
```

Programma 5-2a:

```
00006000:                 DATA      EQU      $6000
00004000:                 PROGRAM   EQU      $4000
                          ;
                                    ORG      DATA
00006000:                 LENGTH    DS.W     1              NUMERO DEI DATI
00006002:                 START     DS.L     1              INDIRIZZO DI BASE DEI DATI
00006006:                 TOTAL     DS.L     1              SOMMA DEGLI ELEMENTI
00010000:                 CARRYBIT  EQU      $10000         VALORE DEL BIT DI CARRY
                          ;
                                    ORG      PROGRAM

00004000: 2078 6002       PGM_5_2A  MOVEA.L  START,A0       INIZIALIZZA IL PUNTATORE
00004004: 7000                      MOVEQ    #0,D0          INIZIALIZZA LA SOMMA A 0
00004006: 3238 6000                 MOVE.W   LENGTH,D1      INIZIALIZZA IL CONTATORE
                          ;
0000400A: 670E                      BEQ.S    DONE           SE LENGTH = 0 ALLORA DONE
0000400C: D058            LOOP      ADD.W    (A0)+,D0       AGGIUNGI IL DATO SEGUENTE
0000400E: 6406                      BCC.S    LOOPTEST       SE CARRY = 0 VAI A LOOPTEST
                          ;
00004010: 0680 0001
00004014: 0000                      ADDI.L   #CARRYBIT,D0   ...ALTRIMENTI AGGIUNGI CARRY
                          ;
00004016: 5341            LOOPTEST  SUBQ.W   #1,D1          AGGIORNA IL CONTATORE
00004018: 66F2                      BNE      LOOP           SE NON E' ZERO VAI A LOOP
                          ;
0000401A: 21C0 6006       DONE      MOVE.L   D0,TOTAL       SALVA IL RISULTATO
                          ;
0000401E: 4E75                      RTS

                                    END      PGM_5_2A
```

Diagramma di Flusso 5-2a

Spiegazione del programma 5.2a

È un programma analogo a quello di addizione a 16 bit. Dal momento che, in questo caso, si tratta di una somma a 32 bit, dobbiamo tener conto del riporto generato dall'istruzione ADD. Due nuove istruzioni (BCC e ADDI) controllano la presenza di un eventuale riporto durante l'addizione e, quando questo si verifica, provvedono ad aggiungere al risultato della somma il bit di Carry.

Se il flag di Carry (C) è = 0, l'istruzione BCC provoca un salto alla locazione di memoria LOOPTEST. Perciò, qualora nell'addizione a 16 bit non esista un riporto, il programma evita l'istruzione

che incrementa i 16 bit più significativi del risultato. L'offset relativo per BCC LOOPTEST è:

```
 4016                  4016
-(400E+2)     =       -4010
                      -----
                        06
```

L'offset relativo per BNE LOOP è:

```
 400C                  400C
-(4018+2)     =       -401A
                      -----
                       -OE      =      FFF2
```

L'offset relativo per BEQ DONE è:

```
 (401A                 401A
-(400A+2)     =       -400C
                      -----
                        OE
```

La forma long word dell'istruzione ADD potrebbere rendere ancora più semplice questo programma. Tuttavia, poichè la serie è formata da valori a 16 bit, sarà necessario del lavoro in più per trasformare questi dati in long word. A questo provvede il programma 5-2b.

Programma 5-2b:

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
                                     ORG      DATA
00006000:                  LENGTH    DS.W     1             NUMERO DEI DATI
00006002:                  START     DS.L     1             INDIRIZZO DI BASE DEI DATI
00006006:                  TOTAL     DS.L     1             SOMMA DEGLI ELEMENTI
                           ;
                                     ORG      PROGRAM

00004000: 2078 6002        PGM_5_2B  MOVEA.L  START,A0      INIZIALIZZA IL PUNTATORE
00004004: 7000                       MOVEQ    #0,D0         INIZIALIZZA LA SOMMA A 0
00004006: 2400                       MOVE.L   D0,D2         AZZERA REG. TEMPORANEO
00004008: 3238 6000                  MOVE.W   LENGTH,D1     INIZIALIZZA IL CONTATORE
                           ;
0000400C: 6708                       BEQ.S    DONE          SE LENGTH = 0 ALLORA DONE
0000400E: 3418             LOOP      MOVE.W   (A0)+,D2      D2[15-0] := DATO
00004010: D082                       ADD.L    D2,D0         SOMMA IL DATO
00004012: 5341                       SUBQ.W   #1,D1         AGGIORNA IL CONTATORE
00004014: 66F8                       BNE      LOOP          SE NON E' ZERO VAI A LOOP
                           ;
00004016: 21C0 6006        DONE      MOVE.L   D0,TOTAL      SALVA IL RISULTATO
                           ;
0000401A: 4E75                       RTS

                                     END      PGM_5_2B
```

Spiegazione del programma 5.2b

In fase di inizializzazione, azzeriamo i 16 bit più significativi del registro D2; dal momento che questi bit non cambieranno mai, non sarà necessario azzerarli ogni volta che viene eseguito il ciclo. I valori a 16 bit, prelevati dalla memoria, sono caricati nei 16 bit di ordine basso di D2 ed un'istruzione ADD.L somma il contenuto a 32 bit di D2 al registro D0. Poichè era stato stabilito che doveva trattarsi di numeri privi di segno, i 16 bit di ordine alto resteranno sempre zero.

Non è necessario controllare la presenza di un eventuale riporto, nella sezione di elaborazione, in quanto, in un'operazione a 32 bit, qualunque riporto proveniente dai 16 bit di ordine basso sarà, automaticamente, propagato nella parte di ordine alto di D0. Le modifiche apportate alla sezione di elaborazione hanno ridotto il numero delle istruzioni e, magari, reso più comprensibile il programma. È diminuito anche il numero di byte del loop. Questo contribuisce anche a ridurre il tempo di esecuzione? La sezione di elaborazione del Programma 5-2a richiede 18 o 36 cicli di clock:

```
ADD        8 cicli
BCC       10 cicli   (12 se non viene effettuato il salto)
ADDI.L   (16) cicli  (non sempre eseguiti)
         ----------
         18 (36) cicli (se è usato BCC.S - 18 (32) cicli)
```

La seconda versione richiede 16 cicli di clock:

```
MOVE       8 cicli
ADD.L     (8) cicli
          --------
          16 cicli
```

La seconda versione è più breve e più veloce. Non è, tuttavia, sempre così. Una singola istruzione più potente può richiedere un tempo di esecuzione più lungo rispetto a due o tre istruzioni più semplici che svolgono la stessa funzione. Siete capaci di fare un esempio?

5-3. Conteggio degli elementi negativi

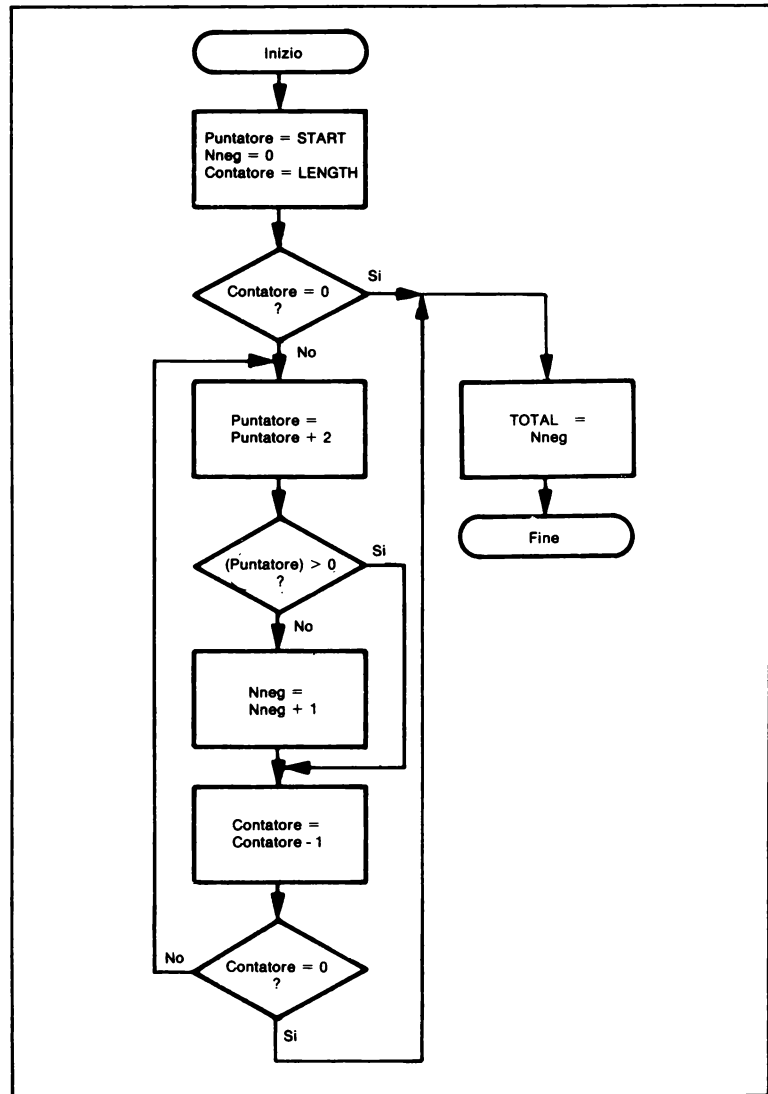
Scopo: Determinare il numero degli elementi negativi presenti in una serie di dati a 16 bit con segno. Gli elementi negativi sono identificati dalla presenza di un 1 nel bit più significativo (bit 15). La lunghezza della serie è definita dalla variabile LENGTH, alla locazione 6000; l'indirizzo iniziale dalla variabile di tipo long word START, alla locazione 6002. Salvare il numero degli elementi negativi nella variabile TOTAL, alla locazione 6006.

Problema Campione:

Input:	LENGTH	– (6000)	=	0003
	START	– (6002)	=	00005000
		(5000)	=	F1DC
		(5002)	=	7E0A
		(5004)	=	824B
Output:	TOTAL	– (6006)	=	0002, poichè le locaz. 5000 e 5004 contengono numeri negativi

Programma 5-3:

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
                                     ORG      DATA
00006000:                  LENGTH    DS.W     1           NUMERO DEI DATI
00006002:                  START     DS.L     1           INDIRIZZO DI BASE DEI DATI
00006006:                  TOTAL     DS.W     1           SOMMA DEGLI ELEMENTI
                           ;
                                     ORG      PROGRAM

00004000: 2078 6002        PGM_5_3   MOVEA.L  START,A0    INIZIALIZZA IL PUNTATORE
00004004: 7000                       MOVEQ    #0,D0       NNEG := 0
00004006: 3238 6000                  MOVE.W   LENGTH,D1   INIZIALIZZA IL CONTATORE
0000400A: 670A                       BEQ.S    DONE        SE LENGTH = 0 VAI A DONE
                           ;
0000400C: 4A58             LOOP      TST.W    (A0)+       CONTROLLA IL DATO
0000400E: 6A02                       BPL.S    LOOPTEST    SE > 0 VAI A LOOPTEST
                           ;
00004010: 5240                       ADDQ.W   #1,D0       ...ALTRIMENTI NNEG:=NNEG+1
                           ;
00004012: 5341             LOOPTEST  SUBQ.W   #1,D1       AGGIORNA IL CONTATORE
00004014: 66F6                       BNE      LOOP        SE NON E' ZERO VAI A LOOP
                           ;
00004016: 31C0 6006        DONE      MOVE.W   D0,TOTAL    SALVA IL NUM. DI ELEM. NEGATIVI
                           ;
0000401A: 4E75                       RTS

                                     END      PGM_5_3
```

Diagramma di Flusso 5-3

Spiegazione del programma 5.3

L'istruzione TST è usata per stabilire se l'elemento successivo della serie è un numero negativo. TST confronta l'operando con zero e modifica i flag di stato di conseguenza. Perciò, il meccanismo dell'istruzione TST è sostanzialmente equivalente a:

SUBQ #0,(A0) +

Perchè, in un caso come questo, si preferisce utilizzare TST invece di SUBQ? Perchè risulta più comprensibile.

Mentre fa il test dell'operando, TST modifica i flag di stato in base al risultato del confronto. I flag di Carry (C) e di Overflow (O) vengono sempre azzerati.

Il flag di Negativo (N) riflette semplicemente il valore del bit 15 del risultato più recente. Se usate numeri provvisti di segno, il bit 15 rappresenta, appunto, il segno (0 se positivo, 1 se negativo); i mnemonici Branch if Plus (BPL) e Branch if Minus (BMI) presuppongono l'impiego di numeri con segno. Si può, comunque, usare il bit 15 per altri scopi, ad es. per verificare lo stato delle periferiche o altri dati ad 1 bit. Anche in questi casi, è possibile controllare il bit 15 con BMI (bit 15 = 1) o BPL (bit 15 = 0); l'istruzione funziona sebbene i mnemonici non abbiano più molto senso. Un elaboratore effettua le sue operazioni senza preoccuparsi del fatto che abbiano più o meno significato per l'utente. L'interpretazione dei risultati è un problema che riguarda il programmatore e non il computer.

I numeri negativi con segno hanno tutti il bit più significativo posto a 1 e sono effettivamente più grandi, se considerati come privi di segno, rispetto a dei valori positivi.

Nel Programma 5-3, l'istruzione BPL (Branch if Plus) provoca una diramazione se il flag di Negativo è 0. Quale altra istruzione di salto potrebbe essere impiegata al posto di BPL?

Potremmo sostituire:

```
TST  (A0)+
BPL  LOOPTEST
```

con

```
MOVE  (A0)+,D3
BTST  #15,D3
BEQ   LOOPTEST
```

L'istruzione BTST controlla un determinato bit dell'operando destinazione. Se il bit è zero, il flag di Zero (Z) viene posto a 1; se il bit è uno, il flag di Zero (Z) è messo a 0. Questa istruzione è molto utile per controllare i bit diversi dal bit di segno; ad esempio, quando è necessario verificare lo stato di un dispositivo periferico. Sebbene l'istruzione BTST consenta di controllare direttamente il contenuto della memoria, in questo modo possono essere controllati soltanto i bit contenuti in un singolo byte. Sareste in grado di riscrivere il Programma 5-3, in modo che BTST controlli il byte più significativo di un elemento a 16 bit presente in memoria?

5-4. Trovare il valore più grande

Scopo: Trovare l'elemento più grande in una serie di numeri binari a 16 bit privi di segno. La lunghezza della serie è definita dalla variabile LENGTH, alla locazione 6000, e l'indirizzo iniziale della serie dalla variabile long word START, alla locazione 6002. Salvare il risultato (l'elemento più

grande privo di segno) nella variabile MAXNUM, alla locazione 6006.

Problema Campione:

Input:	LENGTH	– (6000)	=	0004
	START	– (6002)	=	00005000
		(5000)	=	A48E
		(5002)	=	71AC
		(5004)	=	34F1
		(5006)	=	E57A
Output:	TOTAL	– (6006)	=	E57A, poichè è il maggiore dei quattro numeri privi di segno.

Diagramma di Flusso 5-4a

Programma 5-4a:

```
00006000:                   DATA      EQU      $6000
00004000:                   PROGRAM   EQU      $4000
                            ;
                                      ORG      DATA
00006000:                   LENGTH    DS.W     1              NUMERO DEI DATI
00006002:                   START     DS.L     1              INDIRIZZO DI BASE DEI DATI
00006006:                   MAXNUM    DS.W     1              NUMERO PIU' GRANDE
                            ;
                                      ORG      PROGRAM

00004000: 2078 6002         PGM_5_4A  MOVEA.L  START,A0       INIZIALIZZA IL PUNTATORE
00004004: 7000                        MOVEQ    #0,D0          MAX := 0
00004006: 3238 6000                   MOVE.W   LENGTH,D1      INIZIALIZZA IL CONTATORE
                            ;
0000400A: 670C                        BEQ.S    DONE           SE LENGTH = 0 VAI A DONE
                            ;
0000400C: 3418              LOOP      MOVE.W   (A0)+,D2       TEMP := DATO SUCCESSIVO
0000400E: B042                        CMP.W    D2,D0          CONFRONTA TEMP CON MAX (MAX-TEMP)
00004010: 6402                        BCC.S    LOOPTEST       SE MAX > 0 = TEMP VAI A LOOPTEST
                            ;
00004012: 3002                        MOVE.W   D2,D0          ...ALTRIMENTI MAX := TEMP
                            ;
00004014: 5341              LOOPTEST  SUBQ.W   #1,D1          AGGIORNA IL CONTATORE
00004016: 66F4                        BNE      LOOP           SE NON E' ZERO VAI A LOOP
                            ;
00004018: 31C0 6006         DONE      MOVE.W   D0,MAXNUM      SALVA IL VALORE MASSIMO
                            ;
0000401C: 4E75                        RTS

                                      END      PGM_5_4A
```

Spiegazione del programma 5.4a

Le prime tre istruzioni costutuiscono la sezione di inizializzazione.

In questo programma sfruttiamo il fatto che zero è il più piccolo fra i numeri binari privi di segno. Ponendo a zero il valore iniziale stimato di MAXNUM, il programma lo sostituirà con un valore più grande, a meno che tutti gli elementi dell'array non siano zero. MAXNUM resterà zero anche nel caso in cui la serie non contenga nessun elemento.

La sequenza MOVE.W (A0)+,D2 e CMP.W D2,D0 confronta l'elemento successivo della serie con l'attuale valore massimo. L'istruzione CMP interessa i flag di Carry e di Zero, nel modo seguente (TEMP è il valore dell'elemento attuale e MAX è il valore massimo):

Carry = 0 se MAX $\geq$ TEMP (Maggiore o Uguale)
Carry = 1 se MAX < TEMP (Minore)
Zero = 0 se MAX TEMP $\neq$ (Diverso)
Zero = 1 se MAX = TEMP (Uguale)

Il programma usa l'istruzione di salto BCC (Branch if Carry Clear), che effettua il test sia del flag di Carry che del flag di Zero. Se entrambi sono 1, sostituisce il massimo con l'elemento attuale, mediante l'istruzione MOVE.W D2,D0. Al posto di BCC, poteva essere impiegata l'istruzione di salto BHI, che sarebbe stata più comprensibile. Perchè è preferibile l'istruzione BCC?

Il programma non funziona correttamente con i numeri forniti di segno, perchè i numeri negativi risulterebbero tutti più grandi dei numeri positivi. Nel confrontarli, si dovrebbe usare il flag di Segno (Negativo), invece del flag di Carry. Bisogna, tuttavia, considerare il fatto che l'overflow del complemento a due può modificare il segno; cioè, la grandezza del risultato potrebbe determinare overflow nel bit di segno. L'MC68000 ha delle speciali istruzioni (BGT, BGE, BLE e BLT) che effettuano diramazioni, dopo confronti tra valori

provvisti di segno, e tengono conto, automaticamente, dell'eventuale overflow.
Come già abbiamo avuto occasione di constatare, l'MC68000 permette di svolgere alcune operazioni direttamente sulla memoria, senza richiedere l'impiego di un registro dati. Il Programma 5-4b sfrutta questa possibilità per eliminare l'istruzione MOVE.W (A0)+,D2 del Programma 5-4a.

Programma 5-4b

```
00006000:                   DATA      EQU      $6000
00004000:                   PROGRAM   EQU      $4000
                            ;
                                      ORG      DATA
00006000:                   LENGTH    DS.W     1              NUMERO DEI DATI
00006002:                   START     DS.L     1              INDIRIZZO DI BASE DEI DATI
00006006:                   MAXNUM    DS.W     1              NUMERO PIU' GRANDE
                            ;
                                      ORG      PROGRAM

00004000: 2078 6002         PGM_5_4B MOVEA.L  START,A0       INIZIALIZZA IL PUNTATORE
00004004: 7000                        MOVEQ    #0,D0          MAX := 0
00004006: 3238 6000                   MOVE.W   LENGTH,D1      INIZIALIZZA IL CONTATORE
                            ;
0000400A: 670C                        BEQ.S    DONE           SE LENGTH = 0 VAI A DONE
                            ;
0000400C: B058              LOOP      CMP.W    (A0)+,D0       CONFRONTA IL DATO CON MAX
0000400E: 6404                        BCC.S    LOOPTEST       SE MAX > O = TEMP VAI A LOOPTEST
                            ;
00004010: 3028 FFFE                   MOVE.W   -2(A0),D0      ...ALTRIMENTI MAX := DATO
                            ;
00004014: 5341              LOOPTEST SUBQ.W   #1,D1          AGGIORNA IL CONTATORE
00004016: 66F4                        BNE      LOOP           SE NON E' ZERO VAI A LOOP
                            ;
00004018: 31C0 6006         DONE      MOVE.W   D0,MAXNUM      SALVA IL VALORE MASSIMO
                            ;
0000401C: 4E75                        RTS

                                      END      PGM_5_4B
```

Spiegazione del programma 5.4b

Sebbene possa sembrare che l'istruzione CMP.W (A0)+,D0 semplifichi il programma, essa causa un piccolo problema: incrementa il registro A0, mentre effettua il confronto. Ora, quando deve essere aggiornato il massimo, il nuovo valore non è più contenuto in nessun registro dati ed il suo indirizzo non è presente in alcuno dei registri indirizzi. Per aggirare questa difficoltà, ci possiamo servire dell'indirizzamento indiretto a registro indirizzi con spostamento. Con un valore di spostamento di –2, arretriamo, in sostanza, il puntatore fino all'elemento che abbiamo appena confrontato. L'indirizzo effettivo per l'istruzione MOVE -2(A0),D0 è calcolato nel modo che segue:

Indirizzo Effettivo di –2(A0) = (A0) –2

Il contenuto del registro A0 non viene modificato da questo calcolo.

A prima vista può sembrare che CMP.W (A0)+,D0 non ottimizzi il ciclo di elaborazione, che richiede lo stesso numero di word del Programma 5-4a. Tuttavia, il numero dei cicli di clock necessari per l'esecuzione del Programma 5-4a è di 17 o 20:

MOVE	8	cicli	
CMP	4	cicli	
BCC	5	cicli	(4 se non si verifica il salto)
MOVE	(4)	cicli	(non sempre eseguiti)
	17 (20) cicli		

paragonati ai 13 0 24 cicli del Programma 5-4b:

CMP	8	cicli	
BCC	10	cicli	(4 se non si verifica il salto)
MOVE	(12)	cicli	(non sempre eseguiti)
	13 (24)	cicli	

Sebbene entrambi i programmi richiedano lo stesso numero di cicli per aggiornare il massimo, il secondo è leggermente più efficace quando l'aggiornamento non è necessario.

5-5. Normalizzare un numero binario

Scopo: Effettuare lo shift di un numero binario, finchè il bit più significativo non è 1. L'indirizzo del numero è definito dalla variabile long word NUMBER, alla locazione 6000. Salvare il numero normalizzato (shiftato) nella variabile NORMNUM, alla locazione 6004. Salvare il numero degli shift effettuati nella variabile di un byte SHIFTNUM, alla locazione 6008. Se il numero è zero, azzerare entrambe le variabili NORMNUM e SHIFTNUM.
Il processo di elaborazione è simile a quello necessario per convertire un numero nella notazione scientifica; per esempio:

0.0057 5.7×10^{-3}

Problema Campione:

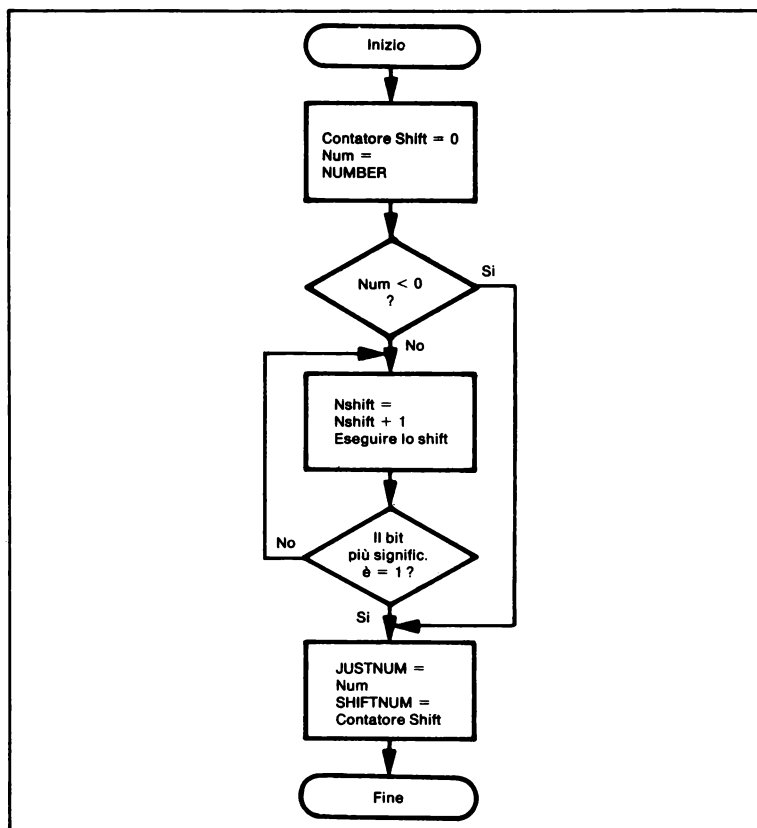
a.	Input:	NUMBER	–	(6000)	=	00005000
				(5000)	=	30001000
	Output:	NORMNUM	–	(6004)	=	C0004000
		SHIFTNUM	–	(6008)	=	02
b.	Input:	NUMBER	–	(6000)	=	00005000
				(5000)	=	00000001
	Output:	NORMNUM	–	(6004)	=	80000000
		SHIFTNUM	–	(6008)	=	1F
c.	Input:	NUMBER	–	(6000)	=	00005000
				(5000)	=	00000000
	Output:	NORMNUM	–	(6004)	=	00000000
		SHIFTNUM	–	(6008)	=	00
d.	Input:	NUMBER	–	(6000)	=	00005000
				(5000)	=	C1234567

Output: NORMNUM – (6004) = C1234567
SHIFTNUM – (6008) = 00

Programma 5-5:

```
00006000:                     DATA      EQU      $6000
00004000:                     PROGRAM   EQU      $4000
                              ;
                                        ORG      DATA
00006000:                     NUMBER    DS.L     1             IND. DEL NUMERO DA NORMALIZZARE
00006004:                     NORMNUM   DS.L     1             NUMERO NORMALIZZATO
00006008:                     SHIFTNUM  DS.B     1             NUMERO DEGLI SHIFT NECESSARI
                              ;
                                        ORG      PROGRAM

00004000: 7000                PGM_5_5   MOVEQ    #0,D0         INIZIALIZZA IL CONTATORE DI SHIFT
00004002: 2078 6000                     MOVEA.L  NUMBER,A0     PRENDI L'INDIRIZZO DEL NUMERO
00004006: 2210                          MOVE.L   (A0),D1       PRENDI IL NUM. DA NORMALIZZARE
00004008: 6F06                          BLE.S    DONE          SE = 0 O NORMALIZZATO VAI A DONE
                              ;
0000400A: 5240                JUSTIFY   ADDQ.W   #1,D0         INCREMENTA IL CONTATORE DI SHIFT
0000400C: E389                          LSL.L    #1,D1         SHIFT DI 1 BIT A SINISTRA
0000400E: 6AFA                          BPL      JUSTIFY       DI NUOVO SE MSB = 0
                              ;
00004010: 11C0 6008           DONE      MOVE.B   D0,SHIFTNUM   SALVA IL CONTATORE DI SHIFT
00004014: 21C1 6004                     MOVE.L   D1,NORMNUM    SALVA IL NUMERO NORMALIZZATO
                              ;
00004018: 4E75                          RTS

                                        END      PGM_5_5
```

Diagramma di Flusso 5-5

Spiegazione del programma 5.5

L'istruzione BLE controlla se il numero è zero o se è già normalizzato. Le condizioni che consentono la diramazione vengono a crearsi durante l'istruzione MOVE, che carica il numero nel registro dati D0. BLE provoca un salto a DONE se il flag di Zero è 1. Se il numero è già normalizzato, il bit più significativo è 1 ed il flag di Negativo verrà posto a 1 da MOVE. In questo caso, BLE causa una diramazione a DONE, se il flag di Negativo è 1. Per quale motivo si può utilizzare BLE per effettuare questo test, dal momento che, quando si fa uso di questa istruzione , bisogna considerare anche lo stato del flag di Overflow (V)?

LSL.L #1,D0 (Logical Shift Left Long) effettua lo shift di un bit verso sinistra del registro dati specificato ed azzera il bit meno significativo. Il bit più significativo finisce nel flag di Carry ed il vecchio valore del Carry viene perduto. L'impiego di LSL equivale a sommare D0 a se stesso; il risultato è, naturalmente, il doppio del valore iniziale.

BPL provoca una diramazione a JUSTIFY, se il flag di Negativo è 0. Questo può significare che il risultato era un numero positivo o, soltanto, che il bit più significativo del risultato era 0; il microprocessore si limita ad effettuare l'operazione: tocca al programmatore fornire l'interpretazione. Dal momento che l'istruzione LSL modifica il flag di Carry, come si potrebbe modificare il programma, utilizzando BCC (Branch if Carry Clear), invece di BPL?

PROBLEMI

5-1. Checksum di dati

Scopo: Calcolare il checksum di una serie di numeri ad 8 bit. La lunghezza della serie è definita dalla variabile LENGTH, alla locazione 6000. L'indirizzo iniziale è contenuto nella variabile long word START, alla locazione 6002. Salvare il checksum nella variabile CHECKSUM, alla locazione 6006. Il checksum è ottenuto mediante l'Or Esclusivo di tutti i valori presenti nella lista.

Nota: I checksum sono spesso usati da sistemi con nastro di carta o registratori a cassette, per accertarsi che i dati siano stati letti correttamente. Il checksum eseguito durante la lettura dei dati viene paragonato ad un valore letto dal nastro insieme ai dati. Se i due valori non sono uguali, il sistema, di solito, indica un errore o rilegge automaticamente i dati.

Problema Campione:

```
Input:   LENGTH      - (6000) = 0003
         START       - (6002) = 00005000
                       (5000) = 28
                       (5001) = 55
                       (5002) = 26
Output:  CHECKSUM    - (6006) = (5000) + (5001) + (5002)
                              = 28 + 55 + 26
                              =   00101000
                              = +01010101
                       ----------------------
                                  01111101
                                 +00100110
                       ----------------------
                                  01011011
                              = 5B
```

5-2. Conteggio di numeri positivi, negativi e zeri

Scopo: Determinare il numero di zeri e di elementi positivi (il bit più significativo è zero, ma l'intero numero è diverso da zero) e negativi (il bit più significativo è 1), in una serie di numeri a 16 bit. La lunghezza della serie è definita dalla variabile LENGTH, alla locazione 6000, e l'indirizzo iniziale della serie è definito dal contenuto della variabile long word START, alla locazione 6002. Mettere il numero degli elementi negativi nella variabile NUMNEG, alla locazione 6006, il numero degli elementi uguali a zero nella variabile NUMZERO, alla locazione 6008, ed il numero di elementi positivi nella variabile NUMPOS, alla locazione 600A.

Problema Campione:

```
Input:   LENGTH      - (6000) = 0006
         START       - (6002) = 00005000
                       (5000) = 7602
                       (5002) = 8D48
                       (5004) = 2120
                       (5006) = 0000
                       (5008) = E605
                       (500A) = 0004
Output: 2 negativi, 1 zero, 3 positivi, perciò

         NUMNEG      - (6006) = 0002
         NUMZERO     - (6008) = 0001
         NUMPOS      - (600A) = 0003
```

5-3. Ricerca del minimo

Scopo: Trovare l'elemento più piccolo in una serie di dati da un byte privi di segno. La lunghezza della serie è definita dalla variabile LENGTH, alla locazione 6000, e l'indirizzo iniziale è contenuto nella variabilelong word START, alla locazione 6002. Mettere il valore minimo nella variabile NUMMIN, alla locazione 6006.

Problema Campione:

```
Input:   LENGTH    - (6000) = 0005
         START     - (6002) = 00005000
                     (5000) = 65
                     (5001) = 79
                     (5002) = 15
                     (5003) = E3
                     (5004) = 72
Output:  NUMMIN    - (6006) = 15, poichè è il minore dei cinque
                     numeri privi di segno
```

5-4. Conteggio dei bit con valore 1

Scopo: Determinare il numero di bit con valore 1 presenti nella variabile a 16 bit NUM, alla locazione 6000, e salvare il risultato nella variabile NUMBITS, alla locazione 6002.

Problema Campione:

```
Input:   NUM       - (6000) = B794 = 1011011110010100
Output:  NUMBITS   - (6002) = 09
```

5-5. Trovare l'elemento con il maggior numero di bit posti a 1

Scopo: Determinare quale elemento, in una serie di numeri a 16 bit, ha il maggior numero di bit con valore 1. La lunghezza della serie è definita dalla variabile LENGTH, alla locazione 6000, e l'indirizzo iniziale è contenuto nella variabile long word START, alla locazione 6002. Salvare il risultato nella variabile NUM, alla locazione 6006. Se due o più elementi hanno lo stesso numero di bit uguali a 1, usare il valore dell'elemento che occupa la posizione più vicina all'inizio della serie.

Problema Campione:

Input:	LENGTH	– (6000)	=	0005		
	START	– (6002)	=	00005000		
		(5000)	=	6779	=	0110011101111001
		(5002)	=	15E3	=	0001010111100011
		(5004)	=	68F2	=	0110100011110010
		(5006)	=	8700	=	1000011100000000
		(5008)	=	592A	=	0101100100101010
Output:	NUM	– (6006)	=	6779, poichè è il primo elemento della serie con dieci bit =1		

CAPITOLO 6

DATI CODIFICATI COME CARATTERI

Il codice ASCII

I microprocessori trattano spesso dati che rappresentano caratteri stampabili, anzichè quantità numeriche. Non solo le tastiere, le telescriventi, i dispositivi di comunicazione, i display ottici e i terminali dei computer si attendono o forniscono dati codificati come caratteri, ma anche molti strumenti e sistemi di collaudo o di controllo richiedono i dati sotto questa forma. Il codice ASCII (American Standard Code for Information Interchange) è quello più diffuso; altri sono il Baudot (telegrafo) e l'EBCDIC (Extended Binary-Coded-Decimal Interchange Code).

In questo volume, per la codifica dei caratteri si abbiamo preferito utilizzare il codice ASCII a 7 bit, mostrato nella Tabella 6-1; il codice del carattere occupa i sette bit di ordine basso di un byte, mentre il bit più significativo contiene uno zero o un bit di parità.

GESTIONE DI DATI IN ASCII

Ordinamento delle stringhe

Ecco alcuni principi da seguire nel trattare dati codificati in ASCII:

1. **I codici per i numeri e le lettere formano delle sequenze ordinate.** Dal momento che i codici ASCII per i numeri da 0 a 9 sono compresi fra 30_{16} e 39_{16} possiamo convertire una cifra decimale nel corrispondente carattere ASCII (e viceversa) mediante la semplice aggiunta di un valore costante: 30_{16} = ASCII 0. Inoltre, dato che i codici per le lettere maiuscole formano una sequenza continua da 41_{16} a $5A_{16}$, per mettere delle stringhe in ordine alfabetico è sufficiente ordinarle in base ai loro valori numerici.
2. **Un computer non è in grado di distinguere tra caratteri stampabili e non.** Solo i dispositivi di I/O fanno una tale distinzione.
3. **Un dispositivo di I/O in ASCII gestisce solamente dati in ASCII.** Ad esempio, volendo stampare la cifra 7 bisognerà inviare ad una stampante ASCII il dato 37_{16}; mentre 07_{16} è il carattere per il campanello. Analogamente, se un operatore preme il tasto 9 su una tastiera ASCII il dato in ingresso sarà 39_{16}; 09_{16} è, infatti, il carattere di tabulazione orizzontale.

Tabella 6-1 - Codici Esadecimali ASCII per i Caratteri

Byte meno significativo \ Byte più significativo	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	−	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Caratteri di controllo

NUL	Carattere Nullo	DC1	Controllo Device 1
SOH	Inizio Intestazione	DC2	Controllo Device 2
STX	Inizio Testo	DC3	Controllo Device 3
ETX	Fine Testo	DC4	Controllo Device 4
EOT	Fine Trasmissione	NAK	Riconoscimento Negativo
ENQ	Enquiry	SYN	Attesa di Sincronizzazione
ACK	Riconoscimento	ETB	Fine Trasmissione di un Blocco
BEL	Campanello	CAN	Cancel
BS	Backspace	EM	End of Medium
HT	Tabulazione Orizzontale	SUB	Sostituzione
LF	Line Feed	ESC	Escape
VT	Tabulazione Verticale	FS	Separatore di File
FF	Form Feed	GS	Separatore di Gruppo
CR	Ritorno Carrello	RS	Separatore di Record
SO	Shift Out	US	Separatore di Unit
SI	Shift In	SP	Spazio
DLE	Data Link Escape	DEL	Delete

4. **Molti dispositivi ASCII non usano l'intero set di caratteri.** Ad esempio, alcuni ignorano molti dei caratteri di controllo oppure non stampano le lettere minuscole.
5. **I caratteri di controllo ASCII spesso vengono interpretati in modo molto diverso.** Ciascun dispositvo ASCII , normalmente, utilizza i caratteri di controllo in un modo particolare allo scopo di permettere, ad esempio, lo spostamento del cursore su un CRT o di gestire via software caratteristiche come la velocità di trasmissione dei dati, la larghezza della stampa e la lunghezza di riga.
6. **Alcuni fra i caratteri di controllo ASCII più largamente impiegati sono:**

   $0A_{16}$ LF (nuova riga)
   $0D_{16}$ CR (ritorno carrello)
   08_{16} arretramento (backspace)
   $7F_{16}$ cancella (delete)

7. **Ogni carattere ASCII occupa otto bit.** Questo consente di disporre di un numero di caratteri molto vasto, ma è uno spreco quando ne vengono utilizzati solo alcuni. Se, ad esempio, un dato consiste interamente di numeri decimali, il formato ASCII (una sola cifra per byte) richiede uno spazio, una capacità di comunicazione ed un tempo di eleborazione doppi rispetto al formato BCD (che consente di avere due cifre per ogni byte).

Molti linguaggi assembly hanno delle caratteristiche che facilitano il trattamento di dati codificati come caratteri. Nel linguaggio assembly della Motorola gli apostrofi che precedono e seguono un

carattere ne indicano il corrispondente valore ASCII. Ad esempio

```
MOVE.B   #'A',D0
```

è uguale a

```
MOVE.B   #$41,D0
```

La prima forma è preferibile per diversi motivi. Innanzitutto aumenta la leggibilità dell'istruzione e, inoltre, evita gli errori derivanti dalla ricerca di un valore in una tabella. Va detto, anche, che il programma, in questo modo, non è vincolato all'ASCII come set di caratteri, poichè l'assemblatore gestisce la conversione, usando qualsiasi codice gli è stato assegnato in fase di progettazione.

ESEMPI DI PROGRAMMAZIONE

6-1. Lunghezza di una stringa di caratteri

Scopo: Determinare la lunghezza di una stringa di caratteri. L'indirizzo iniziale è contenuto nella variabile a 32 bit START, alla locazione 6000. La fine della stringa è indicata da un carattere di ritorno carrello (carriage return) in ASCII ($0D_{16}$). Mettere la lunghezza della stringa (escluso il ritorno carrello) nella variabile LENGTH, alla locazione 6004.

Problemi Campione:

```
a. Input:   START    - (6000) = 00005000
                       (5000) = 0D
   Output:  LENGTH   - (6004) = 0000
b. Input:   START    - (6000) = 00005000
                       (5000) = 4D 'M'
                       (5001) = 43 'C'
                       (5002) = 36 '6'
                       (5003) = 38 '8'
                       (5004) = 30 '0'
                       (5005) = 30 '0'
                       (5006) = 30 '0'
                       (5007) = 0D CR
   Output:  LENGTH   - (6004) = 07
```

Diagramma di Flusso 6-1a

Programma 6-1a:

```
00006000:                 DATA      EQU      $6000
00004000:                 PROGRAM   EQU      $4000
                          ;
                                    ORG      DATA
00006000:                 START     DS.L     1           INDIRIZZO DELLA STRINGA
00006004:                 LENGTH    DS.W     1           NUM. DI CARATTERI NELLA STRINGA
                          ;
0000000D:                 CR        EQU      $0D         COD. ASCII DI RITORNO CARRELLO
                          ;
                                    ORG      PROGRAM

00004000: 2078 6000       PGM_6_1A  MOVEA.L  START,A0    PUNTATORE INIZIO STRINGA
00004004: 7000                      MOVEQ    #0,D0       INIZIALIZZA CONT. LUNGHEZZA
                          ;
00004006: 0C18 000D       LOOP      CMPI.B   #CR,(A0)+   E' UN RITORNO CARRELLO?
0000400A: 6704                      BEQ.S    DONE        SE SI' VAI A DONE
                          ;
0000400C: 5240                      ADDQ.W   #1,D0       ...ALTRIMENTI INCREMENTA CONTATORE
0000400E: 60F6                      BRA      LOOP        CONTINUA LA RICERCA
                          ;
00004010: 31C0 6004       DONE      MOVE.W   D0,LENGTH   SALVA LUNGHEZZA STRINGA
                          ;
00004014: 4E75                      RTS

                                    END      PGM_6_1A
```

Spiegazione del programma 6-1a

Per il processore il ritorno di carrello (CR) è soltanto un altro codice ASCII ($0D_{16}$). Anche se esso costringe un dispositivo d'uscita a compiere una funzione di controllo, anzichè stampare un simbolo, il processore considera $0D_{16}$ semplicemente come uno dei valori da ricercare, al pari degli altri.

La ricerca viene effettuata mediante l'istruzione di confronto CMPI, che modifica i flag, come se l'operando immediato, il carattere di ritorno carrello ($0D_{16}$), fosse stato sottratto dall'operando destinazione. Quest'ultimo (che è il successivo carattere presente nella stringa) resta inalterato. In questo programma, l'istruzione CMPI modifica il flag di Zero nel modo seguente:

Z = 1 se il carattere della stringa è un ritorno carrello
Z = 0 se non è un ritorno carrello

Oltre ad effettuare il confronto, l'istruzione CMPI utilizza anche l'indirizzamento con postincremento per aggiornare il puntatore della stringa di caratteri. In questo modo si è completata la parte riguardante il controllo del ciclo, mostrata nel Diagramma di Flusso 6-1a. La combinazione di parecchie istruzioni come questa, di solito, rende un programma più efficiente. Quali modifiche sarebbe, tuttavia, necessario apportare al diagramma di flusso ed al programma, dovendo salvare anche il puntatore al ritorno carrello?

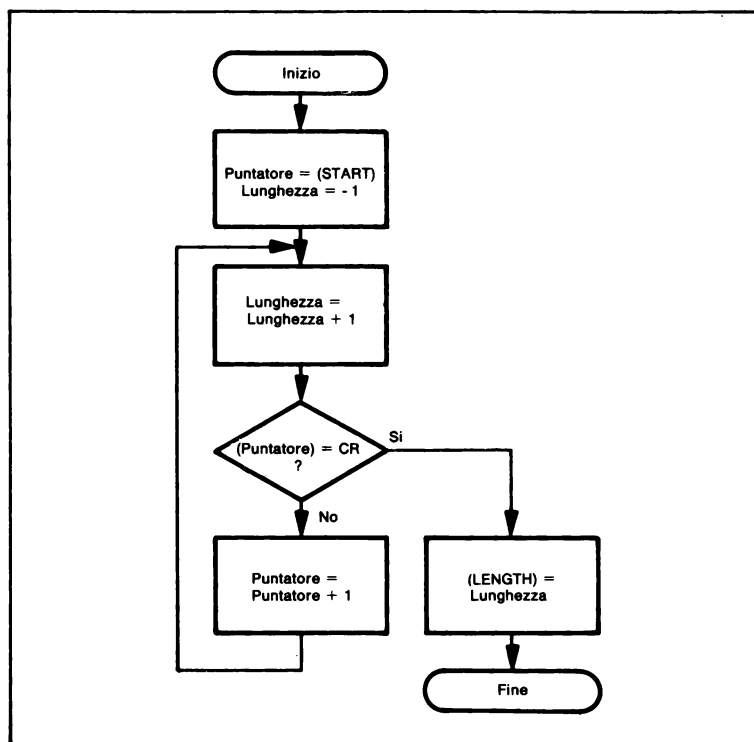
Quello con postincremento è un altro degli indirizzamenti indiretti a registro indirizzi dell'MC68000 in cui ci serviamo del contenuto del registro indicato per calcolare l'indirizzo dell'operando. Una volta che questo è stato utilizzato il processore provvede ad aggiornare il contenuto del registro, incrementandolo dell'opportuno valore (cioè, di uno, due o quattro byte, a seconda che ci riferiamo a dati della grandezza, rispettivamente, di un byte, di una word o di una long word). La sola eccezione si verifica quando viene impiegato il registro indirizzi A7 (Il puntatore allo stack) e il dato ha la dimensione di un byte. In questo caso il puntatore allo stack viene incrementato ugualmente di due byte per essere certi che sia allineato con l'inizio della word successiva.

L'istruzione ADDQ aggiunge 1 al contatore relativo alla lunghezza della stringa, nel registro dati D0. Questo contatore viene azzerato, prima dell'inizio del loop mediante l'istruzione MOVEQ #0,D0. Bisogna ricordarsi di inizializzare le variabili prima di servirsene all'interno di un loop. La mancata inizializzaione è uno degli errori di programmazione più comuni.

Variando la logica e modificando le condizioni iniziali si può accorciare il programma e ridurre il tempo di esecuzione. Se cambiamo il diagramma di flusso in modo che il programma incrementi la lunghezza della stringa prima di verificare la presenza del ritorno carrello, è sufficiente una sola istruzione di salto, invece di due.

Programma 6-1b:

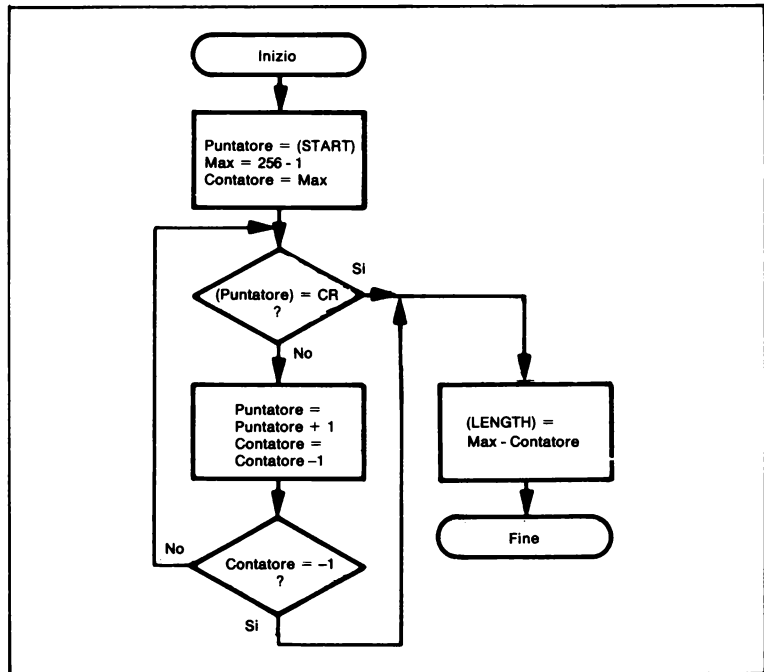
```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
                                  ORG      DATA
00006000:                START    DS.L     1              INDIRIZZO DELLA STRINGA
00006004:                LENGTH   DS.W     1              NUM. DI CARATTERI NELLA STRINGA
                         ;
0000000D:                CR       EQU      $0D            COD. ASCII DI RITORNO CARRELLO
                         ;
                                  ORG      PROGRAM

00004000: 2078 6000      PGM_6_1B MOVEA.L  START,A0       PUNTATORE INIZIO STRINGA
00004004: 70FF                    MOVEQ    #-1,D0         INIZIALIZZA CONT. LUNGHEZZA
00004006: 720D                    MOVEQ    #CR,D1         INIZ. CON IL COD. ASCII DI CR
                         ;
00004008: 5240           LOOP     ADDQ.W   #1,D0          INCREMENTA CONTATORE
0000400A: B218                    CMP.B    (A0)+,D1       IL CARATTERE ATTUALE E' C.R.?
0000400C: 66FA                    BNE      LOOP           SE NON E',CONTINUA RICERCA
                         ;
0000400E: 31C0 6004               MOVE.W   D0,LENGTH      ...ALTRIMENTI SALVA LUNGHEZZA
                         ;
00004012: 4E75                    RTS

                                  END      PGM_6_1B
```

Diagramma di Flusso 6-1b

Spiegazione del programma 6-1b

Come si può osservare dal Programma 6-1b l'incremento della lunghezza della stringa all'inizio del ciclo, anzichè alla fine di esso, permette di eliminare una delle istruzioni di salto. Abbiamo introdotto anche un'altra modifica meno evidente, ma che riduce ulteriormente il tempo di esecuzione: abbiamo usato l'indirizzamento diretto a registro dati per l'operando sorgente dell'istruzione Compare, invece del dato immediato come nel Programma 6-1a. Questo riduce di due byte il codice oggetto dell'istruzione Compare ed evita che il microprocessore debba ricaricare il valore ASCII di ritorno carrello ogni volta che ripete il ciclo. Come regola generale, l'eliminazione di operandi immediati, all'interno di un loop, ne migliora l'efficienza. Il gruppo delle istruzioni "veloci", come MOVEQ e ADDQ, rappresenta un'eccezione. Bisogna anche riconoscere che l'impiego degli operandi immediati rende un programma più comprensibile.

Nessuno dei programmi precedenti ha dei cicli che terminano decrementando a zero un contatore o incrementandolo fino a raggiungere un valore massimo. In realtà, il processore continua semplicemente ad esaminare i caratteri, finchè non trova un ritorno carrello. Evidentemente, questo rappresenterà un problema se la stringa, per un errore o una dimenticanza, non contiene un ritorno carrello. È buona abitudine mettere sempre un contatore anche se, a

prima vista, può sembrare inutile. Cosa accadrebbe se i programmi che vi abbiamo appena presentato fossero usati con una stringa, che non contiene un ritorno carrello? Il Programma 6-1c tiene conto di questa eventualità.

Diagramma di Fusso 6-1c

Programma 6-1c:

```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
                                  ORG      DATA
00006000:                START    DS.L     1              INDIRIZZO DELLA STRINGA
00006004:                LENGTH   DS.W     1              NUM. DI CARATTERI NELLA STRINGA
                         ;
0000000D:                CR       EQU      $0D            COD. ASCII DI RITORNO CARRELLO
                         ;
                                  ORG      PROGRAM

00004000: 2078 6000      PGM_6_1C MOVEA.L  START,A0       PUNTATORE INIZIO STRINGA
00004004: 74FF                    MOVEQ    #256-1,D2      LUNGH. MAX. DELLA STRINGA=256
00004006: 3002                    MOVE.W   D2,D0          CONTATORE := LUNGH. MAX.
00004008: 720D                    MOVEQ    #CR,D1         INIZ. CON IL COD. ASCII DI CR
                         ;
                         *  CERCA UN RITORNO CARRELLO. TERMINA QUANDO E' STATO
                         *  TROVATO OPPURE SONO STATI ESAMINATI 256 CARATTERI
                         ;
0000400A: B218           LOOP     CMP.B    (A0)+,D1       E' UN RITORNO CARRELLO?
0000400C: 57C8 FFFC               DBEQ     D0,LOOP        SE NON E' E NON E' LA FINE
                         ;                                DELLA STRINGA, CONTINUA
00004010: 9440                    SUB.W    D0,D2          DETERMINA LUNGH. STRINGA
00004012: 31C2 6004               MOVE.W   D2,LENGTH      SALVA LUNGH. STRINGA
                         ;
00004016: 4E75                    RTS

                                  END      PGM_6_1C
```

Spiegazione del programma 6-1c

Questo programma utilizza una delle istruzioni di Test. Decremento e Salto: DBcc. Questo gruppo di istruzioni si rivela molto utile in un loop o nell'elaborazione di array. Le istruzioni DBcc hanno la forma

DBcc Dn, <label>

e svolgono le seguenti funzioni:

1. Se la condizione viene soddisfatta il controllo passa all'istruzione successiva a DBcc.
2. Se la condizione non è soddisfatta allora
   a. I 16 bit di ordine basso del registro dati specificato sono decrementati di uno.
   b. Se il risulato è -1, il controllo passa all'istruzione successiva a DBcc.
   c. Se il risultato non è -1 si verifica una diramazione alla locazione di salto specificata che deve trovarsi in un'area di memoria indirizzabile mediante un offset a 16 bit con estensione del segno, relativo al valore attuale del P.C.

I test consentiti dall'istruzione DBcc sono identici ai test delle istruzioni Bcc, tranne per il fatto che DBcc permette anche le condizioni "mai vero" o "falso" (F) e "sempre vero" (T). L'assemblatore della Motorola prevede sia DBRA che DBF.

Con l'istruzione DBEQ la sequenza delle due istruzioni CMP e DBEQ ricerca un carattere di ritorno carrello in una stringa della lunghezza massima di 256 byte. La ricerca termina quando viene incontrato un ritorno carrello oppure quando sono stati valutati tutti i 256 caratteri della stringa. In entrambi i casi, il controllo passa all'istruzione immediatamente successiva a DBEQ. In questo programma viene sempre eseguito lo stesso calcolo, indipendemente dalla causa che pone fine alla ricerca. Tuttavia, possiamo anche effettuare operazioni diverse, a seconda del motivo che ha causato l'uscita dal ciclo. In tal caso, bisogna far seguire all'istruzione DBcc una istruzione di salto Bcc, in modo da trasferire il controllo a quella parte del programma associato con la particolare condizione che ha determinato la fine del ciclo.

Sintassi e funzioni delle istruzioni DBcc

Usando le istruzioni DBcc è indispensabile una corretta inizializzazione dei contatori. Nel Programma 6-1c, il contatore è stato inizializzato a 256-1 (255) poichè il ciclo termina quando il contatore raggiunge -1 e non zero. Si è preferito la forma 256-1, anzichè 255, unicamente per motivi di chiarezza.

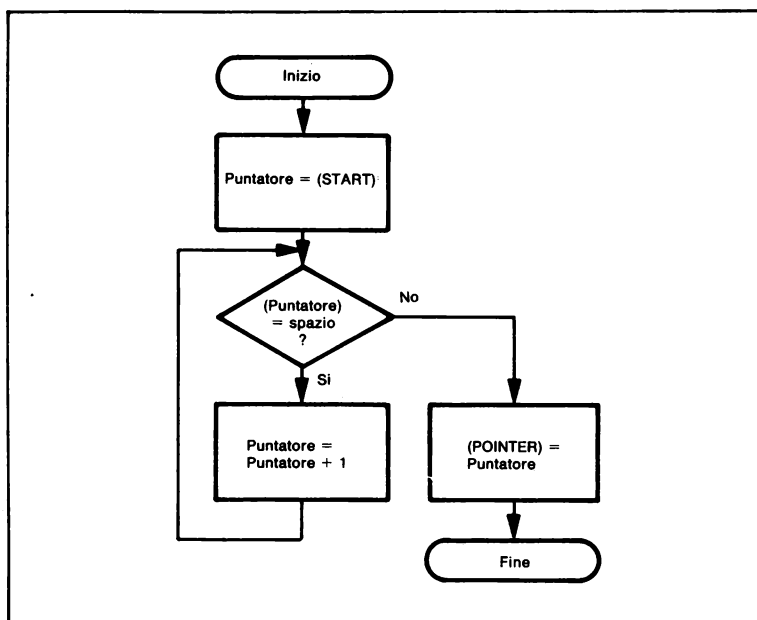
Una volta terminato il ciclo il contatore non contiene la lunghezza della stringa: dobbiamo calcolarla sottraendo il contemuto del contatore dalla lunghezza massima della stringa meno 1. (Ricordatevi della condizione di uscita!).

6-2. Ricerca del primo carattere diverso dallo spazio

Scopo: Cercare in una stringa di caratteri ASCII un carattere diverso dallo spazio. L'indirizzo iniziale della stringa è contenuto nella variabile a 32 bit START, alla locazione 6000. Salvare l'indirizzo del primo carattere diverso dallo spazio nella variabile a 32 bit POINTER, alla locazione 6004. Il carattere di spazio (o blank) corrisponde al codice ASCII 2016.

Problemi Campione:

a.	Input:	START	– (6000)	=	00005000
			(5000)	=	37 '7'
	Output:	POINTER	– (6004)	=	00005000
b.	Input:	START	– (6000)	=	00005000
			(5000)	=	20 spazio
			(5001)	=	20 spazio
			(5002)	=	20 spazio
			(5003)	=	46 'F'
			(5004)	=	20 spazio
	Output:	POINTER	– (6004)	=	00005003, poichè tutte le locazioni di memoria precedenti contenevano dei caratteri di spazio.

Diagramma di Flusso 6-2

Programma 6-2

```
00006000:                   DATA     EQU      $6000
00004000:                   PROGRAM  EQU      $4000
                            ;
                                     ORG      DATA
00006000:                   START    DS.L     1              INDIRIZZO DELLA STRINGA
00006004:                   POINTER  DS.L     1              IND. DEL PRIMO CARATT.DIVERSO DALLO
                            ;                                                             'SPAZIO
00000020:                   SPAZIO   EQU      ' '            CODICE ASCII DELLO SPAZIO
                            ;
                                     ORG      PROGRAM

00004000: 2078 6000         PGM_6_2  MOVEA.L  START,A0       PUNTATORE INIZIO STRINGA
00004004: 7220                       MOVEQ    #SPAZIO,D1     INIZ. CON IL COD. ASCII DI ' '
                            ;
00004006: B218              LOOP     CMP.B    (A0)+,D1       E' UN CARATTERE DI SPAZIO?
00004008: 67FC                       BEQ      LOOP           SE SI' CONTINUA RICERCA
                            ;
0000400A: 5388                       SUBQ.L   #1,A0          ...ALTRIMENTI POSIZIONA IL PUNT.
                            ;                                SU QUESTO CARATTERE
0000400C: 21C8 6004                  MOVEA.L  A0,POINTER     SALVA L'IND. DEL PRIMO
                            ;                                CARATTERE DIVERSO DALLO SPAZIO
00004010: 4E75                       RTS

                                     END      PGM_6_2
```

Spiegazione del programma 6-2

Avrete notato la presenza degli apostrofi (') o virgolette singole prima e dopo ogni carattere ASCII. Per indicare un singolo carattere ASCII in un programma in linguaggio assembly, è necessario appunto farlo precedere e seguire da un apostrofo ('), come, ad es., in uno statement EQU. Come ricorderete l'EQU non è un'istruzione, ma una direttiva destinata all'assemblatore che assegna l'espressione presente nel campo dell'operando alla relativa label. Per mettere il codice ASCII nel byte di ordine basso, è necessario il suffisso .B; altrimenti l'assemblatore mette il valore ASCII nel byte di odine alto di un valore a 16 bit, azzerando i bit restanti.

Per inserire in memoria una stringa di caratteri ASCII bisogna far ricorso alla direttiva DC (Define Constant). Anche in questo caso, la stringa racchiusa fra due apostrofi viene messa nel campo operando di DC. Se all'interno della stringa compare un apostrofo, deve essere preceduto da un altro apostrofo. Ecco alcuni esempi di definizione di una stringa:

```
DC 'ABCD'   Definisci la stringa ABCD
DC 'IT''S'  Definisci la stringa IT'S
```

Ciascun carattere ASCII richiede otto bit, quattro in più rispetto ad una cifra BCD. Il formato ASCII risulta, dunque, inefficiente quando si tratta di memorizzare o trasmettere dati numerici.

Inserimento in memoria di una stringa di caratteri

La ricerca degli spazi in una stringa è un evento presente in gran parte delle applicazioni di un microprocessore. Molti programmi riducono la quantità di memoria occupata rimuovendo i caratteri di spazio che servono solo per migliorare la leggibilità o per ottenere formati particolari. È evidente che salvare o trasmettere dei caratteri di spazio inutili finisce per rappresentare uno spreco di memoria, un sovvraccarico dei mezzi di comunicazione e un aumento del tempo di elaborazione. Gli operatori, tuttavia, trovano più facile inserire dei dati o dei programmi, quando il computer accetta anche degli

spazi in più; si parla, in questo caso, di formato libero, contrapposto al formato fisso. Uno dei compiti di un elaboratore è proprio quello di convertire dati e comandi da formati di facile comprensione per un essere umano a formati che risultano più efficienti per i computer ed i sistemi di comunicazione.

L'indirizzamento con autoincremento, usato nell'istruzione CMP (A0)+,D1 costituisce un modo piuttosto semplice di passare al carattere successivo. Una volta trovato il primo carattere diverso dallo spazio non dobbiamo, però, dimenticarci che il puntatore è già stato incrementato oltre l'indirizzo che vogliamo salvare. Dobbiamo, quindi, togliere l'incremento con l'istruzione SUBQ #1,A0. Questo non sarebbe necessario se ci spostassimo indietro, anzichè in avanti, poichè l'MC68000 autodecrementa un indirizzo prima di usarlo. Come già abbiamo sottolineato, impiegando l'autodecremento l'indirizzo di partenza deve essere maggiore di uno rispetto alla fine della stringa.

6-3. Sostituire gli zeri iniziali con degli spazi

Scopo: Eseguire l'editing di una stringa di caratteri decimali ASCII, sostituendo tutti gli zeri iniziali con degli spazi. L'indirizzo iniziale della stringa è contenuto nella variabile long word START, alla locazione 6000. I primi due byte della stringa ne rappresentano la lunghezza in byte. I caratteri veri e propri cominciano con il terzo byte.

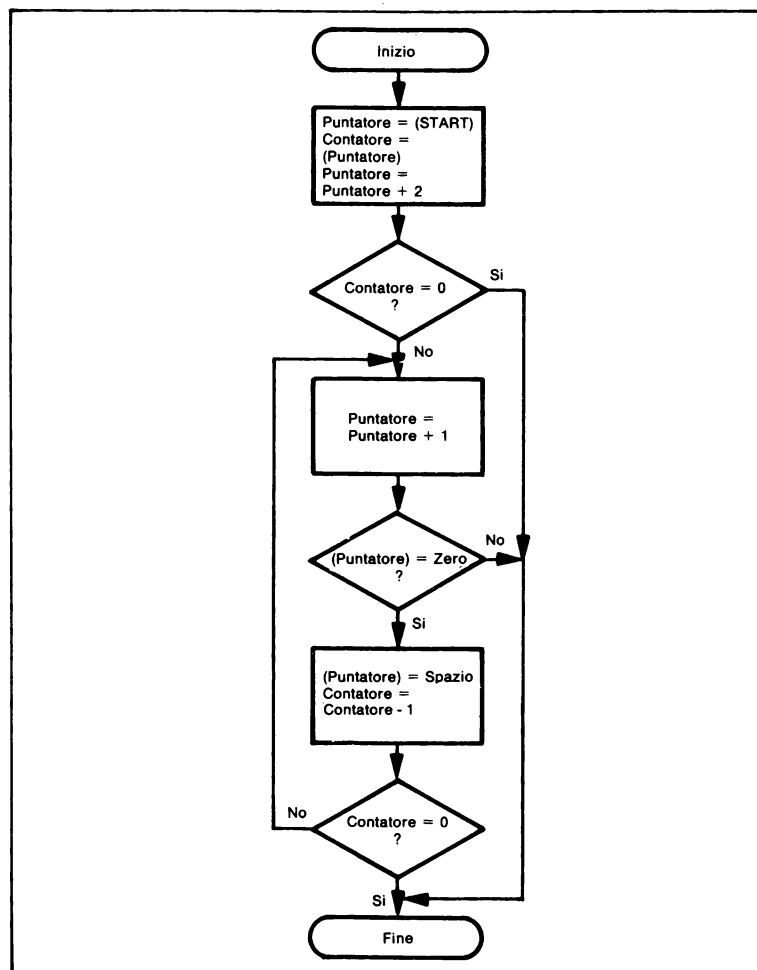
Problemi Campione:

```
a. Input:   START   - (6000) = 00005000
                      (5000) = 0002 Lung. della stringa in byte
                      (5002) = 36 '6'
                      (5003) = 39 '9'
```

Il programma lascia la stringa invariata, poichè la cifra iniziale non è zero.

```
a. Input:   START   - (6000) = 00005000
                      (5000) = 0002 Lung. della stringa in
                      byte
                      (5002) = 30 '0'
                      (5003) = 30 '0'
                      (5004) = 38 '8'
                      (5002) = 20 spazio
                      (5003) = 20 spazio
                      (5004) = 38 '8'
```

Il programma sostituisce i due zeri inziali con caratteri ASCII di spazio. Verrebbe stampato '8...', anzichè '008...'.

Diagramma di Flusso 6-3

Programma 6-3

```
00006000:                   DATA     EQU      $6000
00004000:                   PROGRAM  EQU      $4000
                            ;
                                     ORG      DATA
00006000:                   START    DS.L     1              INDIRIZZO DELLA STRINGA
                            ;
00000030:                   CHAR_0   EQU      '0'            COD. ASCII PER ZERO
00000020:                   SPAZIO   EQU      ' '            COD. ASCII DI SPAZIO
                            ;
                                     ORG      PROGRAM

00004000: 2078 6000         PGM_6_3  MOVEA.L  START,A0       PUNTATORE INIZIO STRINGA
00004004: 7030                       MOVEQ    #CHAR_0,D0     INIZ. CON IL COD. ASCII DI ZERO
00004006: 7220                       MOVEQ    #SPAZIO,D1     INIZ. CON IL COD. ASCII DI SPAZIO
00004008: 3418                       MOVE.W   (A0)+,D2       LUNGH. DELLA STRINGA IN D2
0000400A: 670E                       BEQ.S    DONE           SE LUNGH.= 0 VAI A DONE
0000400C: 5342                       SUBQ.W   #1,D2          AGGIUSTA IL CONTATORE PER DBRA
                            ;
0000400E: B018              LOOP     CMP.B    (A0)+,D0       E' UNO ZERO?
00004010: 6608                       BNE.S    DONE           SE NON E' VAI A DONE
                            ;
00004012: 1141 FFFF                  MOVE.B   D1,-1(A0)      SOSTITUISCI ZERO CON SPAZIO
00004016: 51CA FFF6                  DBRA     D2,LOOP        TERMINA SE TUTTI I CARATT.='0'
                            ;
0000401A:                   DONE     EQU      *
                            ;
0000401A: 4E75                       RTS
                                     END      PGM_6_3
```

Spiegazione del programma 6-3

Questo tipo di formato della stringa, con la lunghezza posta all'inizio, è usato molto spesso nelle applicazioni dei microprocessori. In questo modo possiamo conoscere la lunghezza, senza dover cercare un ritorno di carrello e spostare le stringhe in memoria con più facilità.

Molto spesso capita di dover fare l'editing di stringhe di decimali, per migliorarne l'apparenza. Le procedure tipiche prevedono la rimozione degli zeri iniziali, la giustificazione, l'aggiunta di segni, di delimitatori o di simboli indicanti l'unità di misura (come $, % o #) e l'arrotondamento. I programmi dovrebbero stampare i numeri nella forma che l'utente si aspetta; risultati del tipo "0006", "$27.3482" o "135000000" confondono e spesso risultano del tutto incomprensibili.

Il loop di questo programma ha due condizioni di uscita: una quando il processore trova una cifra diversa da zero e l'altra quando ha terminato la ricerca dell'intera stringa. In una applicazione reale bisogna aver cura di lasciare uno zero, qualora tutte le cifre della stringa siano zero. Sareste in grado di modificare il programma per ottenere questo ?

Siamo partiti dal presupposto che tutte le cifre della stringa siano in codice ASCII; cioè, si sono usati valori compresi tra 30_{16} e 39_{16}, invece della rappresentazione binaria dei numeri da 0 a 9. Per convertire una cifra da BCD ad ASCII è sufficiente sommare 30_{16} (zero in ASCII), mentre per convertire da ASCII a decimale basta sottrarre lo stesso valore.

L'istruzione MOVE.B D1,-1(A0) mette un carattere ASCII di spazio nella locazione di memoria che prima conteneva uno zero, sempre in ASCII.Viene usato l'indirizzamento indiretto a registro indirizzi con uno spostamento di -1 per compensare il + 1 sommato al registro A0 dall'istruzione CMP.B (A0)+,D0.

L'istruzione DBRA fa in modo che il programma non continui oltre la fine della stringa. DBRA è una variante dell'istruzione DBcc, per la quale la condizione di test non è mai vera. DBRA, o la forma equivalente DBF, corrispondono alla sequanza di istruzioni:

```
 SUBI.W #1,D2
 BNE    LOOP
```

L'istruzione DBRA causa sempre un ritorno a LOOP a meno che non sia stata esaminata l'intera stringa (D2 =-1).

6-4. Aggiunta della parità pari ai caratteri ASCII

Utilizzo del bit di parità

Scopo: Aggiungere il bit di parità di tipo pari ad una stringa di caratteri ASCII a 7 bit. L'indirizzo iniziale della stringa è contenuto nella long word START, alla locazione 6000. La prima word della stringa ne indica la lunghezza in byte. I caratteri veri e propri cominciano con il terzo byte. Il bit di parità è il bit più significativo di un byte che, nel caso della

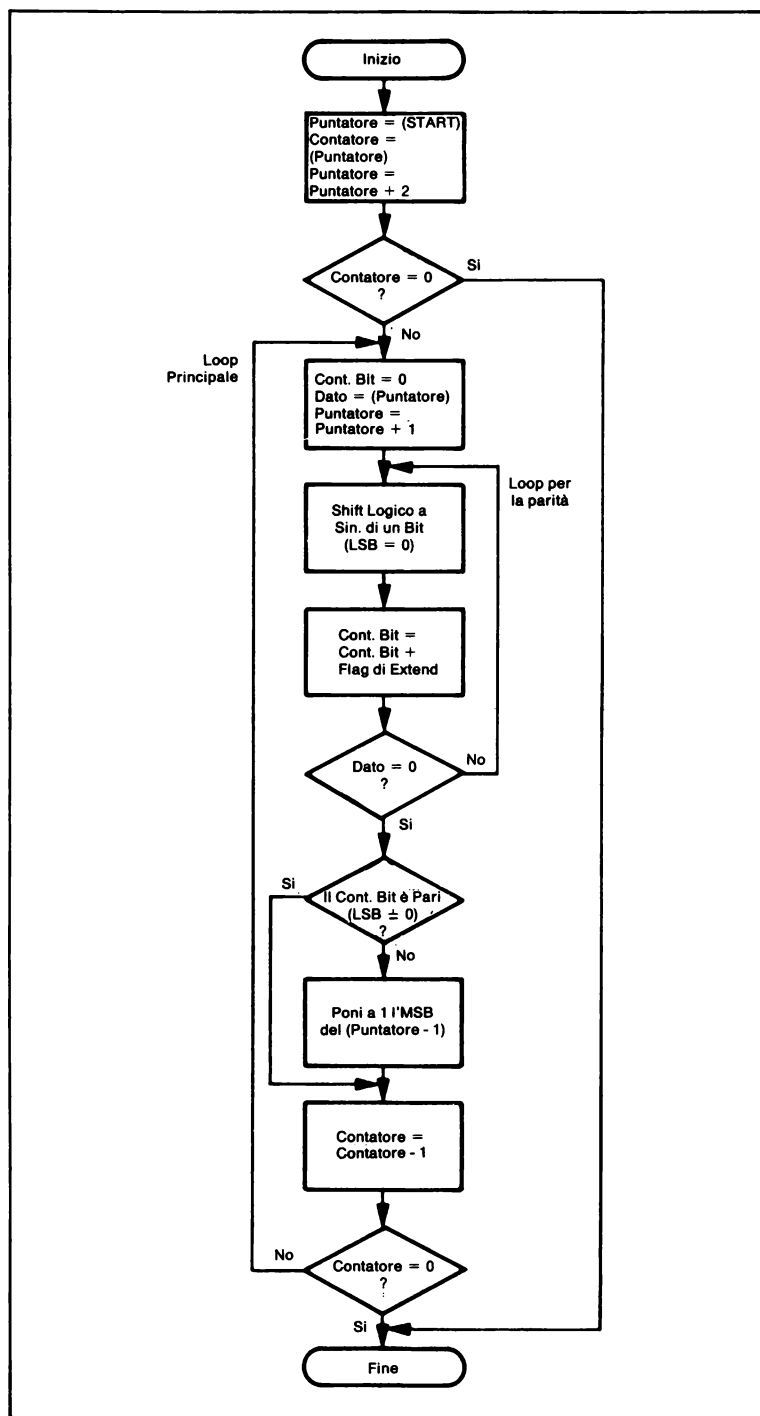
parità pari, viene posto a 1 se ciò fa in modo che il totale di bit uguali a 1 presenti nel byte sia un numero pari; altrimenti è azzerato. In entrambi i casi il numero finale di bit 1 è pari.

Problema Campione:

Input:	START	– (6000)	=	00005000	
		(5000)	=	0006	lungh. della stringa
		(5002)	=	31	0011 0001
		(5003)	=	32	0011 0010
		(5004)	=	33	0011 0011
		(5005)	=	34	0011 0100
		(5006)	=	35	0011 0101
		(5007)	=	36	0011 0110
Output:		(5002)	=	B1	1011 0001
		(5003)	=	B2	1011 0010
		(5004)	=	33	0011 0011
		(5005)	=	B4	1011 0100
		(5006)	=	35	0011 0101
		(5007)	=	36	0011 0110

Programma 6-4:

```
00006000:                 DATA     EQU      $6000
00004000:                 PROGRAM  EQU      $4000
                          ;
                                   ORG      DATA
00006000:                 START    DS.L     1              INDIRIZZO DELLA STRINGA
                          ;
                                   ORG      PROGRAM

00004000: 2078 6000       PGM_6_4  MOVEA.L  START,A0       PUNTATORE INIZIO STRINGA
00004004: 3418                     MOVE.W   (A0)+,D2       LUNGH. STRINGA IN D2
00004006: 6720                     BEQ.S    DONE           SE LUNGH. = 0 VAI A DONE
00004008: 5342                     SUBQ.W   #1,D2          AGGIUSTA CONTATORE PER DBRA
0000400A: 7600                     MOVEQ    #0,D3          COST. 0 PER L'ISTR. ADDX
                          ;
0000400C:                 MAIN_LOOP EQU     *
0000400C: 1218                     MOVE.B   (A0)+,D1       PRENDI IL CARATTERE
0000400E: 7000                     MOVEQ    #0,D0          AZZERA IL CONTATORE DEI BIT
                          ;
00004010:                 PARITY_LOOP EQU   *
00004010: E309                     LSL.B    #1,D1          SHIFT MSB NEI FLAG C E X
00004012: D103                     ADDX.B   D3,D0          SOMMA IL FLAG X AL CONT. DI BIT
00004014: 4A01                     TST.B    D1             CONTATI TUTTI I BIT = 1?
00004016: 66F8                     BNE      PARITY_LOOP    NO? ALLORA CONTINUA
                          ;
00004018: 0800 0000                BTST     #0,D0          ...ALTRIMENTI CONTROLLA SE
                          ;                                LA PARITA' E' DISPARI
0000401C: 6706                     BEQ.S    NEXT_CHAR      SE E' PARI PASSA AL PROSSIMO CAR.
                          ;
0000401E: 08E8 0007
00004022: FFFF                     BSET     #7,-1(A0)      ...ALTRIMENTI PONI A 1 L'MSB
                          ;
00004024:                 NEXT_CHAR EQU     *
00004024: 51CA FFE6                DBRA     D2,MAIN_LOOP   SE CI SONO ALTRI CAR. CONTINUA
                          ;
00004028:                 DONE     EQU      *              ORA LA PARITA' E' PARI
                          ;
00004028: 4E75                     RTS

                                   END      PGM_6_4
```

Diagramma di Flusso 6-4

La parità rappresenta un metodo facile per rilevare eventuali errori su linee di trasmissione disturbate. Se un trasmettitore invia, insieme con i dati, anche la parità, il ricevitore può controllare la correttezza dei dati ricevuti e, in caso di errore, chiedere un nuovo invio. Con un solo bit sbagliato la parità non sarà rispettata poichè il numero di bit uguali a 1 del dato sarà chiaramente cambiato da dispari a pari o viceversa. Tuttavia, se i bit errati sono due risulterà una parità identica a quella del dato originale. Ne deriva che **con il metodo della parità possiamo individuare errori di un bit, ma non quelli di due bit.** Il test mantiene, comunque, la sua validità, poichè sono molto più frequenti i casi di errori singoli.

Un problema più grave, a proposito della **parità,** sta nel fatto che essa **non consente di correggere gli errori.** Qualunque sia la posizione in cui l'errore si verifica, provoca sempre lo stesso cambiamento di parità, così il ricevitore non può stabilire qual è il bit sbagliato. **Tecniche di codifica più avanzate oltre ad individuare un errore danno anche la possibilità di correggerlo. La parità, in ogni modo, è facile da calcolare e si dimostra ancora valida nei casi in cui dover ritrasmettere il dato non costituisce un grosso problema.**

La procedura per il calcolo della parità consiste nel contare il numero di bit uguali a 1 presenti in ciascun byte. Se il numero è dispari e si desidera una parità di tipo pari, il programma pone a 1 il bit più significativo (MSB). Uno dei vantaggi del codice ASCII a 7 bit è quello di lasciare il bit più significativo disponibile per la parità, al contrario del codice EBCDIC ad 8 bit.

L'istruzione LSL azzera il bit meno significativo del registro dati o della locazione di memoria su cui effettua lo shift. Perciò, **una serie di istruzioni LSL porterà in definitiva ad un valore zero indipendentemente dal dato di partenza.** Provate voi stessi! La procedura per il conteggio dei bit, nel nostro esempio, non si serve di un contatore per uscire dal ciclo, ma termina non appena sono rimasti soltanto bit uguali a 0. Questo metodo ha il vantaggio di essere semplice e di ridurre, in molti casi, il tempo di esecuzione.

Spiegazione del programma 6-4

Il Programma 6-4 parte dal presupposto che il bit più significativo (il bit di parità) di ogni dato ad 8 bit inizialmente sia zero. Se fosse 1, allora il Programma 6-4 finirebbe per generare una parità dispari, invece di una pari.

Oltre ad azzerare il bit meno significativo del dato, l'istruzione LSL modifica i flag di Carry (C) e di Extend (E) nel modo seguente:

C = X = 1 se il MSB del dato = 1 prima dello shift
C = X = 0 se il MSB del dato = 0 prima dello shift

Lo stato del flag di Extend è impiegato nell'istruzione ADDX.B D3,D0 che ha la stessa funzione di:

$$DO = D0 + D3 + X = D0 + 0 + X = D0 + X$$

Perciò il numero dei bit uguali a 1 si trova nel registro D0.

Come le altre istruzioni Add, ADDX modifica i flag di stato per cui e necessario usare l'istruzione TST per stabilire se l'istruzione LSL ha azzerato il registro dati. TST.B D1 confronta il contenuto del byte di ordine basso del registro D1 con zero e modifica i flag di stato di conseguenza, senza alterare il contenuto del registro. L'istruzione TST è una forma ottimizzata dell'istruzione di confronto immediato CMPI #0,D1.

Istruzioni per la Manipolazione dei Bit

Significato delle istruzioni BCLR, BCHG, BSET, BTST

L'MC68000 consente operazioni sui singoli bit presenti *in un byte o in una long word.* L'istruzione Bit Clear (BCLR) serve ad azzerare un singolo bit, mentre Bit Change (BCHG) ne inverte il valore. L'istruzione Bit Set (BSET) è impiegata per porre a 1 un determinato bit. Infine, è disponibile l'istruzione Bit Test (BTST) per controllare lo stato di un bit senza modificarlo. Tutte queste istruzioni effettuano un Bit Test implicito (BTST) prima di agire sul bit indicato.

6-5. Confronto con una stringa campione

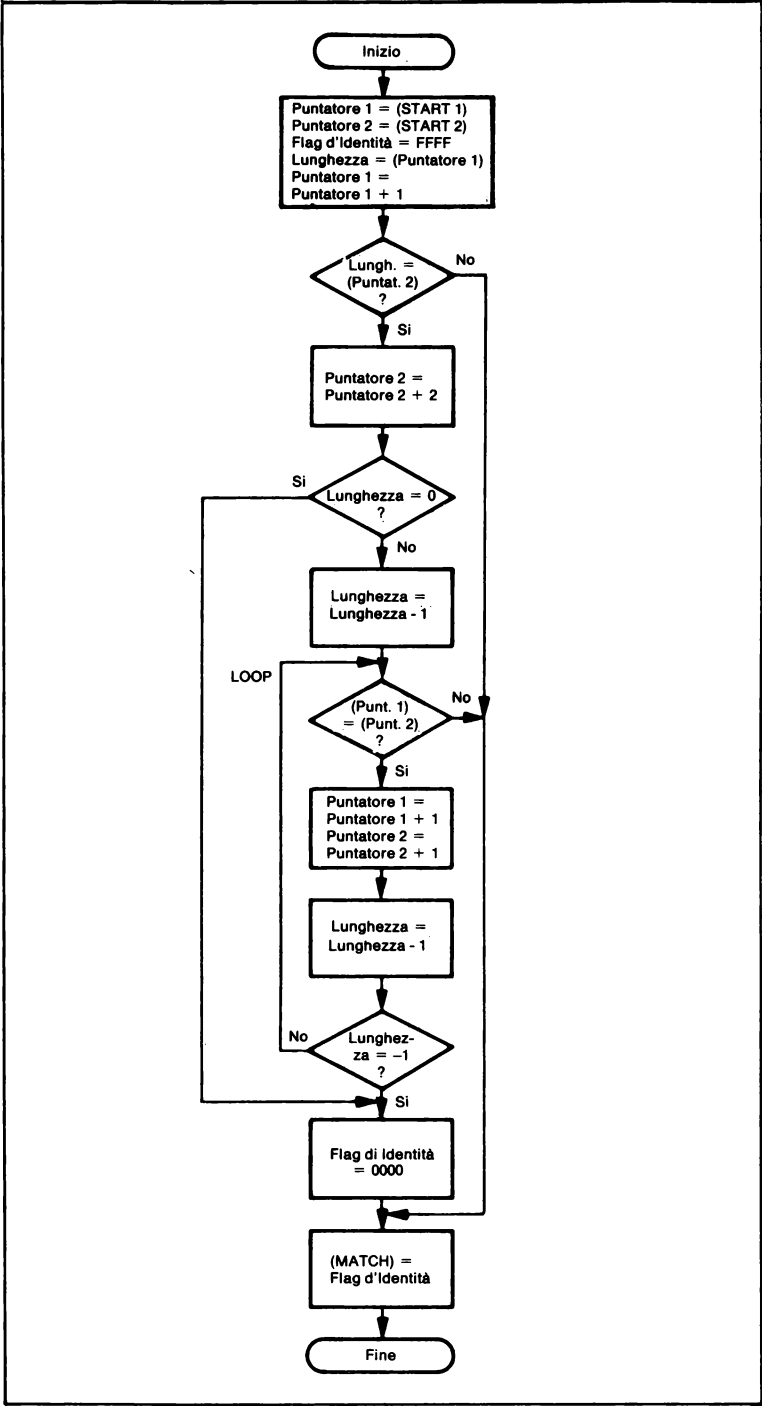
Scopo: Confrontare due stringhe di caratteri ASCII e stabilire se sono uguali. Gli indirizzi iniziali delle stringhe sono contenuti nelle variabili long word START1, alla locazione 6000, e START2, alla locazione 6004. Il primo byte di ciascuna stringa ne contiene la lunghezza (in byte) ed è seguito dalla stringa vera e propria. Se le due stringhe sono uguali, azzerare la variabile MATCH, alla locazione 6008; altrimenti mettere il suo valore a -1 (tutti uno = $FFFF_{16}$).

Problemi Campione:

```
a. Input:   START1   - (6000) = 00005000
            START2   - (6004) = 00005400
                       (5000) = 03
                       (5001) = 43 'C'
                       (5002) = 41 'A'
                       (5003) = 54 'T'
                       (5400) = 03
                       (5401) = 43 'C'
                       (5402) = 41 'A'
                       (5403) = 54 'T'

   Output:  MATCH    - (6004) = 0000 0, poichè le stringhe sono
                       uguali
```

```
b. Input:   START1    - (6000) = 00005000
            START2    - (6004) = 00005400
                        (5000) = 03
                        (5001) = 43 'C'
                        (5002) = 41 'A'
                        (5003) = 54 'T'
                        (5400) = 03
                        (5401) = 52 'R'
                        (5402) = 41 'A'
                        (5403) = 54 'T'

   Output:  MATCH     - (6004) = FFFF -1, poichè i primi caratteri
                        sono diversi

c. Input:   START1    - (6000) = 00005000
            START2    - (6004) = 00005400
                        (5000) = 03
                        (5400) = 04

   Output:  MATCH     - (6004) = FFFF -1, poichè le stringhe han-
                        no lunghezza diversa
```

Nota: il confronto termina non appena è stata trovata la differenza. Il resto della stringa non viene esaminato.

Diagramma di Flusso 6-5a

Programma 6-5a

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
                                     ORG      DATA
00006000:                  START1    DS.L     1               IND. DELLA PRIMA STRINGA
00006004:                  START2    DS.L     1               IND. DELLA SECONDA STRINGA
00006008:                  MATCH     DS.W     1               FLAG D'IDENTITA'
                           ;
                                     ORG      PROGRAM

00004000: 2078 6000        PGM_6_5A MOVEA.L  START1,A0        PUNTATORE PRIMA STRINGA
00004004: 2278 6004                 MOVEA.L  START2,A1        PUNTATORE SECONDA STRINGA
00004008: 72FF                      MOVEQ    #-1,D1           SI PRESUMONO DIVERSE
0000400A: 7000                      MOVEQ    #0,D0            CONT. LUNGH. := 0
0000400C: 1018                      MOVE.B   (A0)+,D0         INIZ. CONTATORE LUNGH.
0000400E: B019                      CMP.B    (A1)+,D0         LUNGHEZZE UGUALI?
00004010: 6610                      BNE.S    DONE             NO, ALLORA SONO DIVERSE
                           ;
00004012: 4A00                      TST.B    D0               LUNGH. STRINGA = 0?
00004014: 670A                      BEQ.S    SAME             SE = 0 ALLORA STRINGHE UGUALI
                           ;
00004016: 5340                      SUBQ.W   #1,D0            AGGIUSTA IL CONTAT. PER DBNE
                           ;
00004018: B308             LOOP     CMPM.B   (A0)+,(A1)+      CONFRONTA I CARATTERI
0000401A: 56C8 FFFC                 DBNE     D0,LOOP          SE SONO UGUALI E LA STRINGA
                           ;                                  NON E' FINITA, CONTINUA
0000401E: 6602                      BNE.S    DONE             SE DIVERSI E FINE STRINGA->DONE
                           ;
00004020: 4641             SAME     NOT.W    D1               LE STRINGHE SONO UGUALI
00004022: 31C1 6008        DONE     MOVE.W   D1,MATCH         SALVA IL RISULTATO
                           ;
00004026: 4E75                      RTS

                                    END      PGM_6_5A
```

Spiegazione del programma 6.5a

Il confronto di stringhe di caratteri ASCII è una parte essenziale del riconoscimento dei nomi e dei comandi, dell'identificazione delle variabili o dei codici operativi negli assemblatori e nei compilatori, dell'accesso ai file, ecc.

L'istruzione MOVEQ #-1,D1 serve a presupporre che non vi sia identità. Se, invece, questa si verifica, il flag di identità viene azzerato dall'istruzione NOT.W D1, che complementa lo stato di ogni bit dell'operando destinazione, per cui un bit zero diventa 1 e viceversa. Se non avessimo inizializzato il flag in questo modo, la fine del programma sarebbe risultata più complessa:

```
         BNE    DONE
SAME:    MOVE   #-1, MATCH
         BRA    DONE
FINI:    MOVE   #0, MATCH
DONE     RTS
```

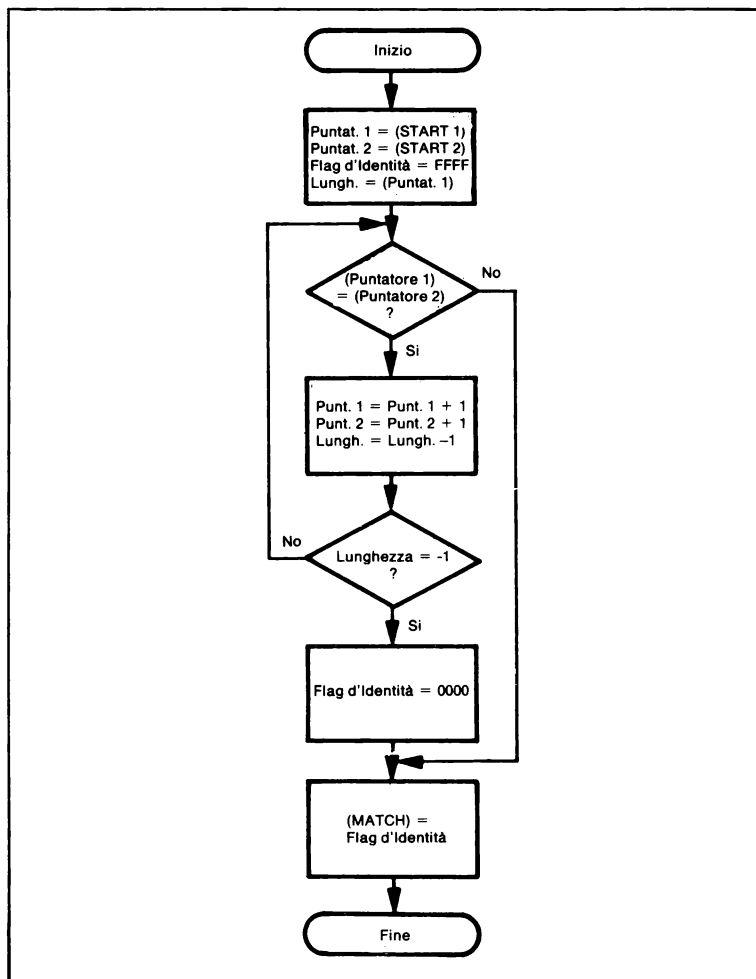
Il metodo di partire dal presupposto che un risultato sia vero finchè non è stato dimostrato il contrario, o viceversa, è una tecnica molto diffusa, che semplifica molti programmi.

L'istruzione CMPM (Compare Memory) permette di confrontare direttamente i dati in memoria, senza doverne prima spostare uno in un registro dati, ed è, quindi, estremamente utile per eseguire confronti fra stringhe. Con questa istruzione è possibile unicamente l'indirizzamento con postincremento per indicare gli operandi. Naturalmente, questo è il modo più utile per confrontare delle stringhe, poichè gli indirizzi sono automaticamente incrementati così da puntare al successivo elemento che dobbiamo confrontare.

Quando il controllo passa all'istruzione successiva a DBNE sappiamo che due caratteri sono diversi oppure che le stringhe sono identiche. L'istruzione BNE serve a stabilire qual è la condizione che ha determinato l'uscita da DBNE. Il corretto funzionamento dell'istruzione BNE deriva dal fatto che DBNE non modifica i flag del registro di stato.

Perchè deve essere usata l'istruzione MOVEQ #0,D0 prima di caricare il byte di ordine basso del registro D0 con la lunghezza della stringa?

Questo programma è molto più complesso del necessario. In realtà, potremmo considerare i byte con le lunghezze delle stringhe come se ne fossero parte integrante; perciò, se non sono uguali, anche le stringhe saranno diverse.

Diagramma di Flusso 6-5b

Programma 6-5b:

```
00006000:                   DATA      EQU      $6000
00004000:                   PROGRAM   EQU      $4000
                            ;
                                      ORG      DATA
00006000:                   START1    DS.L     1              IND. DELLA PRIMA STRINGA
00006004:                   START2    DS.L     1              IND. DELLA SECONDA STRINGA
00006008:                   MATCH     DS.W     1              FLAG D'IDENTITA'
                            ;
                                      ORG      PROGRAM

00004000: 2078 6000         PGM_6_5B MOVEA.L START1,A0        PUNTATORE PRIMA STRINGA
00004004: 2278 6004                  MOVEA.L START2,A1        PUNTATORE SECONDA STRINGA
00004008: 72FF                       MOVEQ   #-1,D1           SI PRESUMONO DIVERSE
0000400A: 7000                       MOVEQ   #0,D0            CONTAT. LUNGHEZZA := 0
0000400C: 1010                       MOVE.B  (A0),D0          INIZ. CONTAT. LUNGHEZZA
                            ;
0000400E: B308              LOOP      CMPM.B   (A0)+,(A1)+    CONFRONTA I DUE CARATTERI
00004010: 56C8 FFFC                   DBNE     D0,LOOP        SE SONO UGUALI E NON E' LA FINE
                            ;                                 DELLA STRINGA, CONTINUA
00004014: 6602                        BNE.S    DONE           SE DIVERSI E FINE STRINGA->DONE
                            ;
00004016: 4641              SAME      NOT.W    D1             LE STRINGHE SONO UGUALI
                            ;
00004018: 31C1 6008         DONE      MOVE.W   D1,MATCH       SALVA IL RISULTATO
                            ;
0000401C: 4E75                        RTS

                                      END      PGM_6_5B
```

Se la lunghezza delle strighe è diversa, il programma termina dopo la prima iterazione. Perchè è possibile usare la lunghezza della stringa come contatore del ciclo, senza prima decrementarla di 1?

PROBLEMI

6-1. Lunghezza di un messaggio per telescrivente

Scopo: Calcolare la lunghezza di un messaggio ASCII. Si tratta di caratteri ASCII a 7 bit con il MSB = 0. L'indirizzo iniziale della stringa di caratteri contenente il messaggio è nella variabile START, alla locazione 6000. Il messaggio inizia con il carattere ASCII STX (02_{16}) e termina con ETX (03_{16}). Salvare la lunghezza del messaggio (il numero dei caratteri compreso fra STX e ETX, escluso questi due) nella variabile LENGTH, alla locazione 6004.

Problema Campione:

Input:	START	– (6000)	=	00005000
		(5000)	=	02 STX
		(5001)	=	47 'G'
		(5002)	=	4F 'O'
		(5003)	=	03 ETX
Output:	LENGTH	– (6004)	=	02, poichè ci sono due caratteri fra STX alla locazioene 6000 e ETX alla locazione 5003

6-2. Trovare l'ultimo carattere diverso dallo spazio

Scopo: Cercare in una stringa di caratteri ASCII l'ultimo carattere diverso dallo spazio. L'indirizzo iniziale della stringa è contenuto nella variabile START, alla locazione 6000, e la stringa termina con un carattere di ritorno carrello ($0D_{16}$). Mettere l'indirizzo dell'ultimo carattere diverso dallo spazio nella variabile ADDRESS, alla locazione 6004.

Problemi Campione:

```
a. Input:   START    - (6000) = 00005000
                       (5000) = 37 '7'
                       (5001) = 0D CR
   Output:  ADDRESS  - (6004) = 5000
```

Poichè l'ultimo (e unico) carattere diverso dallo spazio è nella locazione di memoria 5000.

```
b. Input:   START    - (6000) = 5000
                       (5000) = 41 'A'
                       (5001) = 20 SP
                       (5002) = 48 'H'
                       (5003) = 41 'A'
                       (5004) = 54 'T'
                       (5005) = 20 SP
                       (5006) = 20 SP
                       (5007) = 0D CR
   Output:  ADDRESS  - (6004) = 5004
```

6-3. Riduzione di una stringa decimale alla forma intera

Scopo: Eseguire l'editing di una stringa di caratteri decimali ASCII, sostituendo tutte le cifre a destra del punto decimale con dei caratteri ASCII di spazio (20_{16}). L'indirizzo iniziale della stringa è contenuto nella variabile START, alla locazione 6000, e si presuppone che la stringa contenga esclusivamente cifre decimali codificate in ASCII ed eventualmente un punto decimale ($2E_{16}$). Salvare la lunghezza della stringa nella variabile LENGTH, alla locazione 6004. Se nella stringa non compare il punto decimale si presuppone che questo si trovi in corrispondenza dell'estremità destra.

Problemi Campione:

```
a. Input:   START     - (6000) = 00005000
            LENGTH    - (6004) = 0004 Lunghezza della stringa
                        (5000) = 37 '7'
                        (5001) = 2E '.'
                        (5002) = 38 '8'
                        (5003) = 31 '1'
   Output:              (5000) = 37 '7'
                        (5001) = 2E '.'
                        (5002) = 20 SP
                        (5003) = 20 SP
b. Input:   START     - (6000) = 00005000
            LENGTH    - (6004) = 0003 Lunghezza della stringa
                        (5000) = 36 '6'
                        (5001) = 37 '7'
                        (5002) = 31 '1'
   Output:      Immutato, perchè il numero è 671.
```

6-4. Controllo della parità di tipo pari e dei caratteri ASCII

Scopo: Controllare la parità di tipo pari in una stringa di caratteri ASCII. L'indirizzo iniziale della stringa è contenuto nella variabile START, alla locazione 6000. Il primo byte della stringa contiene la sua lunghezza ed è seguito dalla stringa vera e propria. Se la parità di tutti i caratteri della stringa è corretta, azzerare la variabile PARITY, alla locazione 6004; altrimenti, mettere tutti uno ($FFFF_{16}$) in PARITY.

Problemi Campione:

```
a. Input:   START     - (6000) = 00005000
                        (5000) = 03 Lunghezza della stringa
                        (5001) = B1 = 1011 0001
                        (5002) = B2 = 1011 0010
                        (5003) = 33 = 0011 0011
   Output:  PARITY    - (6004) = 0000, poichè tutti i caratteri hanno
                        parità pari

b. Input:   START     - (6000) = 00005000
                        (5000) = 03 Lunghezza della stringa
                        (5001) = B1 = 1011 0001
                        (5002) = B6 = 1011 0110
                        (5003) = 33 = 0011 0011
   Output:  PARITY    - (6004) = FFFF, poichè il carattere nella loca-
                        zione di memoria 5002 non ha parità pari
```

6-5. Confronto fra stringhe

Scopo: Confrontare due stringhe di caratteri ASCII e stabilire qual è la più grande (cioè, quale segue l'altra in ordine alfabetico). Entrambe le stringhe hanno la stessa lunghezza, definita dalla variabile LENGTH, alla locazione 6000. Gli indirizzi iniziali delle stringhe sono definiti dalle variabili START1, alla locazione 6002, e START, alla locazione 6006. Se la stringa definita da START1 è maggiore o uguale all'altra, azzerare la variabile GREATER, alla locazione 600A; altrimenti, mettere tutti uno in GREATER ($FFFF_{16}$).

Problemi Campione:

```
a. Input:   LENGTH    - (6000) = 0003 Lungh. dI ogni stringa
            START1    - (6002) = 00005000
            START     - (6006) = 00005400
                        (5000) = 43 'C'
                        (5001) = 41 'A'
                        (5002) = 54 'T'

                        (5400) = 42 'B'
                        (5401) = 41 'A'
                        (5402) = 54 'T'

   Output:  GREATER   - (600A) = 0000, poichè CAT è "maggiore"
                        di BAT

b. Input:   LENGTH    - (6000) = 0003 Lungh. di ogni stringa
            START1    - (6002) = 00005000
            START     - (6006) = 00005400
                        (5000) = 43 'C'
                        (5001) = 41 'A'
                        (5002) = 54 'T'

                        (5400) = 43 'C'
                        (5401) = 41 'A'
                        (5402) = 54 'T'
   Output:  GREATER   - (600A) = 0000, poichè CAT non è "mag-
                        giore" di CAT

c. Input:   LENGTH    - (6000) = 0003 Lungh. di ogni stringa
            START1    - (6002) = 00005000
            START     - (6006) = 00005400
                        (5000) = 43 'C'
                        (5001) = 41 'A'
                        (5002) = 54 'T'

                        (5400) = 43 'C'
                        (5401) = 53 'U'
                        (5402) = 54 'T'
   Output:  GREATER   - (600A) = FFFF, poichè CUT è "maggio-
                        re" di CAT
```

CAPITOLO 7

CONVERSIONE DI CODICE

La conversione di un codice è un problema frequente nelle applicazioni dei microcomputer. Le periferiche forniscono dati in ASCII, BCD o in altri codici speciali, che devono essere convertiti in un formato standard, in modo da poter essere elaborati. I dispositivi di uscita richiedono di solito dati in ASCII, BCD, sette segmenti o in eventuali altri codici. Perciò, il sistema, una volta terminata l'elaborazione, deve convertire i risultati nel formato opportuno.

Esistono diversi modi per eseguire una conversione di codice:

1. **Alcune conversioni possono essere facilmente realizzate mediante algoritmi che prevedono delle funzioni logiche o aritmetiche.** Alcune volte, un programma è costretto a gestire separatamente dei casi speciali.
2. **Conversioni più complesse sono effettuate mediante tabelle di riferimento.** Questo metodo riduce eventuali difficoltà di programmazione ed è facilmente applicabile. Qualora si abbia una vasta gamma di valori d'ingresso, la tabella finisce, tuttavia, per occupare una notevole quantità di memoria.
3. **Alcune conversioni sono eseguite mediante l'impiego di componenti hardware.** Ricordiamo, ad esempio, i decodificatori da BCD a sette segmenti ed i Trasmettitori/Ricevitori Asincroni Universali (UART) per la conversione tra formato parallelo (ASCII) e seriale (telescrivente).

In molte applicazioni è il programma che si incarica del processo di conversione. Questo riduce il numero dei componenti e la dissipazione di potenza, risparmiando spazio sulla scheda ed aumentando l'affidabilità del sistema. Inoltre, gran parte delle conversioni di codice sono piuttosto semplici e comportano tempi brevi in fase di esecuzione.

ESEMPI DI PROGRAMMAZIONE

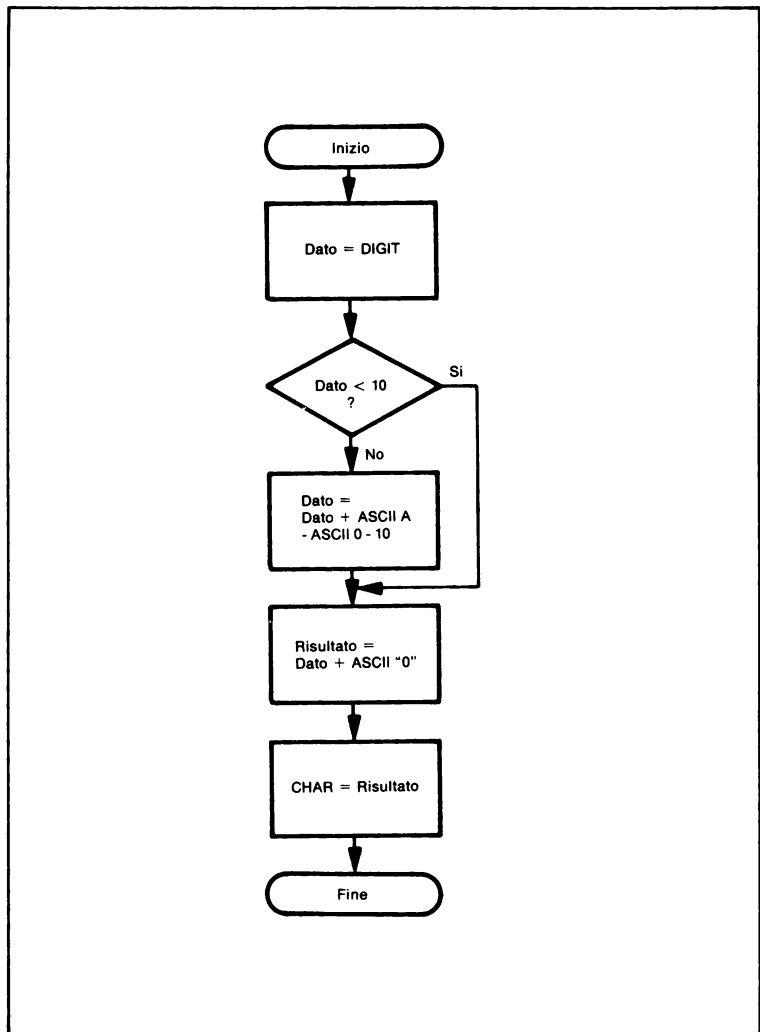
7-1. Da esadecimale ad ASCII

Scopo: Convertire il contenuto della variabile DIGIT, alla locazione 6000, in un carattere ASCII che rappresenti il valore esadecimale della variabile. DIGIT contiene una sola cifra

esadecimale (i quattro bit più significativi sono zero). Salvare il carattere ASCII nella variabile CHAR, alla locazione 6001.

Problemi Campione:

a.	Input:	DIGIT	– (6000)	= OC
	Output:	CHAR	– (6001)	= 43 'C'
b.	Input:	DIGIT	– (6000)	= 06
	Output:	CHAR	– (6001)	= 36 '6'

Diagramma di Flusso 7-1

Programma 7-1

```
00006000:                  DATA     EQU      $6000
00004000:                  PROGRAM  EQU      $4000
                           ;
00006000:                  DIGIT    EQU      $6000           INDIRIZZO DI DIGIT
00006001:                  CHAR     EQU      $6001           INDIRIZZO DI CHAR
                           ;
                                    ORG      PROGRAM

00004000: 1038 6000        PGM_7_1  MOVE.B   DIGIT,D0        PRENDI LA CIFRA ESADECIMALE
00004004: 0C00 000A                 CMP.B    #10,D0          E' < 10?
00004008: 6D02                      BLT.S    ADD_0           SE SI' AGGIUNGI SOLO '0'
                           ;
0000400A: 5E00                      ADD.B    #'A'-'0'-10,D0  ...ALTRIMENTI AGGIUNGI ANCHE
                           ;                                 L'OFFSET PER 'A'-'F'-10
0000400C: 0600 0030        ADD_0    ADD.B    #'0',D0         CONVERTI AD ASCII
00004010: 11C0 6001                 MOVE.B   D0,CHAR         SALVA LA CIFRA IN COD. ASCII
                           ;
00004014: 4E75                      RTS

                                    END      PGM_7_1
```

Spiegazione del programma 7-1

L'idea che sta alla base di questo programma è quella di sommare il valore 0 in ASCII (30_{16}) a tutte le cifre esadecimali. Questa addizione converte in modo corretto le cifre da 0 a 9; tuttavia, le lettere da A a F non seguono immediatamente nel codice ASCII la cifra 9. C'è un' interruzione fra il codice ASCII per 9 (39_{16}) e quello per A (41_{16}) di cui **la conversione deve tener conto, sommando un'ulteriore costante ai valori maggiori di 9** (A, B, C, D e F). A questo provvede la prima istruzione ADD sommando 'A' - '0' - 10 al registro dati D0. Sapete spiegare perchè per le lettere è necessario il valore 'A' - '0' - 10? Notate come per rappresentare questo valore siano sufficienti i 3 bit del campo dati di un'istruzione ADDQ. L'assemblatore se ne accorge e genera, automaticamente, il codice oggetto corrispondente ad ADDQ (anche se il mnemonico d'istruzione è diverso). Come si può costringere l'assemblatore a produrre il codice oggetto relativo ad ADDI?

Nel programma sorgente abbiamo usato la forma ASCII per indicare gli addendi; un apostrofo prima e dopo un carattere ne indica l'equivalente ASCII. L'offset per le lettere è stato lasciato sotto forma di espressione aritmetica, per renderne più chiaro il significato. In questo modo, la struttura del programma risulta più comprensibile, anche se ciò comporta un aumento, del resto trascurabile, del tempo di assemblaggio. Una routine come questa la ritroviamo in molte applicazioni: ad esempio, nei programmi monitor, i quali devono convertire cifre esadecimali nei loro equivalenti ASCII, allo scopo di mostrare il contenuto delle locazioni di memoria su una stampante o un terminale video.

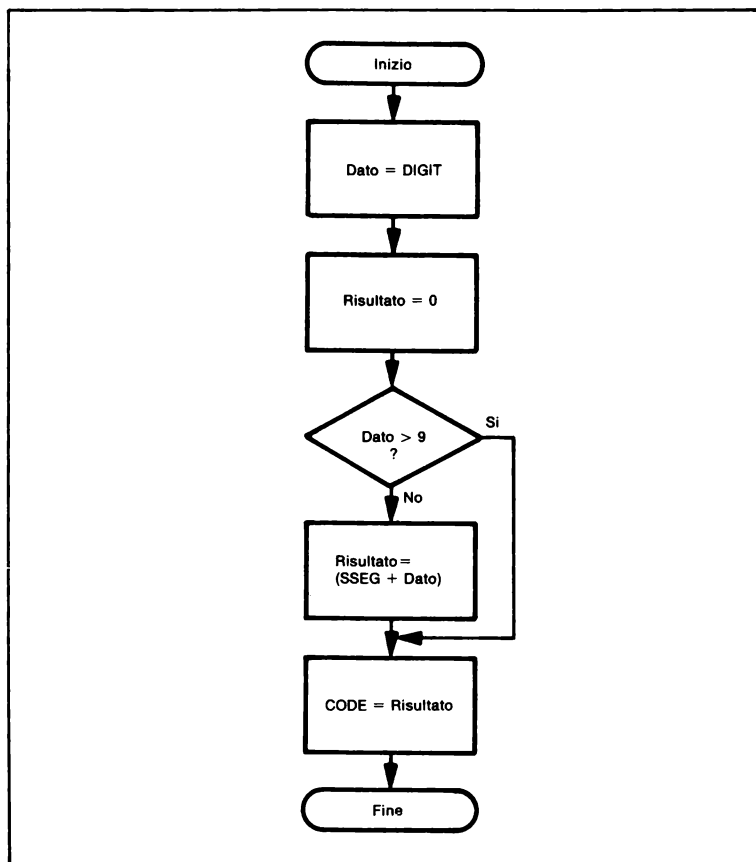
7-2. Da decimale a sette segmenti

Scopo: Convertire il contenuto della variabile DIGIT, alla locazione 6000, in un codice a sette segmenti e salvarlo nella variabile CODE, alla locazione 6001. Se DIGIT non contiene una singola cifra decimale, azzerare CODE.

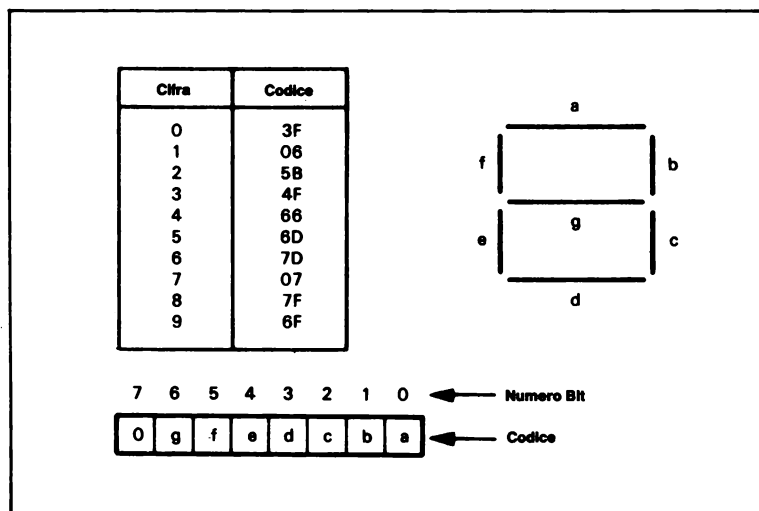
La Figura 7-1 mostra un display a sette segmenti e la relativa rappresentazione in codice binario. Ai segmenti vengono, di solito, assegnate le lettere dalla a alla g, come appare dalla figura. Abbiamo organizzato il codice a sette segmenti nel modo seguente: il segmento g corisponde al bit 6, il segmento f al bit 5 e così via. Il bit 7 è sempre 0. I nomi dei segmenti sono standard ma l'associazione con i vari bit è del tutto arbitraria; nelle applicazioni reali questa corrispondenza è stabilita dall'hardware.

Il codice "sette-segmenti"

Nella Figura 7-1 trovate una tipica tabella per la conversione dei numeri decimali in un codice a sette segmenti; essa utilizza la logica positiva, cioè 1 = acceso e 0 = spento. Osservate come la tabella preveda 7D per 6, anzichè 7C (segmento in alto spento), in modo da evitare confusione con la lettera b minuscola, e 6F per 9, anzichè 67 (segmento in basso spento), per simmetria con il 6.

Diagramma di Flusso 7-2

Si noti come la somma dell'indirizzo di base (SSEG) e dell'indice (DIGIT) dia come risultato l'indirizzo contenente la risposta.

Cifra	Codice
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F

Figura 7-1
Disposizione dei sette segmenti

Problemi Campione:

a Input: DIGIT – (6000) = 03
Output: CHAR – (6001) = 4F

b. Input: DIGIT – (6000) = 28
Output: CHAR – (6001) = 00

c. Input: DIGIT – (6000) = 0A
Output: CHAR – (6001) = 00

Programma 7-2:

```
00006000:                  DATA     EQU      $6000
00004000:                  PROGRAM  EQU      $4000
                           ;
                                    ORG      DATA

00006000:                  DIGIT    DS.B     1                 CIFRA
00006001:                  CODE     DS.B     1                 CODICE BCD
00006002:                  SSEG     DC.B     $3F,$06,$5B,$4F,$66,$6D,$7D,$07,$7F,$6F TABELLA DI
                           ;                                                  CONVERSIONE
                                    ORG      PROGRAM

00004000: 207C 0000
00004004: 6002             PGM_7_2  MOVEA.L  #SSEG,A0          PUNTATORE TABELLA DI CONVERSIONE
00004006: 4201                      CLR.B    D1
00004008: 1038 6000                 MOVE.B   DIGIT,D0          PRENDI CIFRA
0000400C: 0C00 0009                 CMP.B    #9,D0             E' VALIDA?
00004010: 6206                      BHI.S    DONE              SE NON E' VALIDA AZZERA RISULTATO
                           ;
00004012: 4880                      EXT.W    D0                TRASFORMA L'INDICE IN UNA WORD
00004014: 1230 0000                 MOVE.B   0(A0,D0),D1       PRENDI IL CODICE DALLA TABELLA
                           ;
00004018: 11C1 6001        DONE     MOVE.B   D1,CODE           SALVA IL CODICE BCD
                           ;
0000401C: 4E75                      RTS

                                    END      PGM_7_2
```

L'istruzione CLR

L'istruzione Clear (CLR), al pari dell'istruzione MOVEQ+, serve ad azzerare tutti i 32 bit di un registro dati e richiede soltanto una word d'istruzione. Tuttavia CLR, a differenza di MOVEQ+, può anche essere usata per azzerare il byte o la word di ordine basso di un registro dati. (Nel nostro programma ci serviamo appunto di CLR.B D1 per azzerare gli otto bit meno significativi di D1). Inoltre, con CLR possiamo anche azzerare direttamente una locazione di memoria. Per quale ragione l'MC68000 dispone di modi diversi per azzerare la memoria o i registri?

Spiegazione del programma 7-2

Il programma calcola l'indirizzo di memoria del codice a sette segmenti sommando un indice - la cifra da convertire - all'indirizzo di base della tabella. Questa procedura è detta "table lookup". La somma non richiede alcuna istruzione esplicita, dal momento che il processore, con l'indirizzamento indicizzato, la esegue automaticamente durante il calcolo dell'indirizzo effettivo. La tabella può essere collocata in una parte qualsiasi della memoria, poichè per questa somma indicizzata sono utilizzati tutti i 32 bit del registro indirizzi.

Nell'indirizzamento indicizzato, mentre sono impiegati tutti i 32 bit del registro indirizzi primario nella determinazione dell'indirizzo, del registro indice (o registro di offset) viene usata soltanto la word meno significativa. Nel nostro programma il valore di spostamento (o offset) all'interno della tabella ha la grandezza di un byte, per cui il suo caricamento nel registro dati interessa solo gli 8 bit meno significativi e non altera gli altri 24. Comunque sarà necessario azzerare i bit 8-15 di questo registro, per poterlo usare come indice. A questo provvede l'istruzione EXT che trasforma il byte o la word presenti nel registro, rispettivamente, in una word o in una long word, mediante l'estensione del bit più significativo (MSB). Quindi se il bit più significativo è zero tutti i bit a sinistra del dato saranno azzerati; se il bit è 1 verrano posti tutti a 1.

L'Uso della Direttiva Define Constant (DC)

La direttiva dell'assemblatore DC (Define Constant) pone dati costanti, della lunghezza di un byte, nella memoria di programma. Può trattarsi di tabelle, intestazioni, messaggi d'errore, messaggi per la richiesta di input, caratteri per la formattazione di testi, valori soglia e costanti matematiche. La label relativa alla pseudo-operazione DC è associata all'indirizzo della locazione in cui l'assemblatore mette il primo byte dei dati.

L'assemblatore assegna i dati introdotti mediante la direttiva DC ad indirizzi di memoria consecutivi, limitandosi alla loro conversione numerica. Con una singola direttiva DC possiamo inserire un qualsiasi numero di dati, separati da una virgola.

Le tabelle rappresentano un metodo semplice, rapido ed efficace per risolvere i problemi relativi a conversioni di codice più complesse della conversione da esadecimale ad ASCII che vi abbiamo mostra-

to nel nostro esempio. Le varie tabelle devono semplicemente contenere tutti i risultati possibili, disposti in base al valore dell'input; cioè, il primo elemento deve corrispondere al valore di input zero e così via.

I display a sette segmenti riescono a rappresentare in modo riconoscibile le cifre decimali, alcune lettere ed altri caratteri; sono poco costosi e facili da gestire con dei microprocessori. Tuttavia, molti ancora non riescono a leggere bene le cifre codificate a sette segmenti, benchè il diffondersi del loro impiego nei calcolatori e negli orologi le abbia rese molto familiari.

7-3. Da ASCII a decimale

Scopo: Convertire il contenuto della variabile CHAR, alla locazione 6000, da un carattere ASCII a una cifra decimale e salvare il risultato nella variabile DIGIT, alla locazione 6001. Se il contenuto di CHAR non è la rappresentazione ASCII di una cifra decimale, mettere in DIGIT il valore FF_{16}.

Problemi Campione:

a Input: CHAR – (6000) = 37 '7'
Output: DIGIT – (6001) = 07

b. Input: CHAR – (6000) = 55 'U' (codice non valido, perchè non è una cifra decimale ASCII)
Output: DIGIT – (6001) = FF

Diagramma di Flusso 7-3

Programma 7-3:

```
00006000:                DATA     EQU     $6000
00004000:                PROGRAM  EQU     $4000
                         ;
00006000:                DIGIT    EQU     $6000          INDIRIZZO DI DIGIT
00006001:                CHAR     EQU     $6001          INDIRIZZO DI CHAR
                         ;
                                  ORG     PROGRAM

00004000: 72FF           PGM_7_3  MOVEQ   #-1,D1         METTI A -1 IL FLAG D'ERRORE
00004002: 1038 6001               MOVE.B  CHAR,D0        PRENDI IL CARATTERE
00004006: 0400 0030               SUB.B   #'0',D0        E' MINORE DI '0'?
0000400A: 6508                    BCS.S   DONE           SE SI', NON E' UNA CIFRA
                         ;
0000400C: 0C00 0009               CMP.B   #9,D0          E' MAGGIORE DI '9'?
00004010: 6202                    BHI.S   DONE           SE SI' NON E' UNA CIFRA
                         ;
00004012: C141                    EXG     D0,D1          PRENDI IL VAL. NUMERICO DEL CAR.
                         ;
00004014: 11C1 6000      DONE     MOVE.B  D1,DIGIT       SALVA CIFRA O FLAG D'ERRORE
                         ;
00004018: 4E75                    RTS

                                  END     PGM_7_3
```

Spiegazione del programma 7-3

Questo programma gestisce i caratteri codificati in ASCII come dei semplici numeri. Dal momento che l'ASCII assegna una sequenza ordinata di codici alle cifre decimali **riusciamo a stabilire se un carattere ASCII corrisponde ad una cifra verificando se cade all'interno di un certo intervallo di valori numerici.** Allo stesso modo, sempre grazie alla successione ordinata dei codici ASCII, è possibile stabilire se un carattere appartiene o meno ad un particolare gruppo di lettere o simboli, ad esempio da A a F. **Questo metodo presuppone la conoscenza dettagliata di un particolare codice e non è necessariamente valido per altri codici.**

Se dal codice ASCII relativo ad una qualsiasi cifra decimale sottraiamo il codice ASCII corrispondente allo 0 (30_{16}) otterremo il valore decimale di quella cifra. Un carattere ASCII è una cifra decimale se il suo valore è compreso fra 30_{16} e 39_{16}. Come si può stabilire se un carattere ASCII corrisponde ad una delle cifre esadecimali? La conversione da ASCII a decimale è utilizzata nelle applicazioni in cui sono inseriti dei dati decimali mediante un dispositivo ASCII, come una telescrivente o un terminale.

Il programma effettua un confronto (con il più piccolo valore ammesso) mediante una sottrazione (SUB.B '0',D0) necessaria anche alla conversione da ASCII a decimale. Per l'altro confronto ricorre ad una sottrazione implicita (CMP.B #9,D0) in modo da non perdere l'eventuale cifra decimale. **Le sottrazioni implicite (CMP) sono di gran lunga più comuni di quelle reali, in quanto molto spesso non ci interessa il valore numerico risultante.**

L'istruzione EXG

L'istruzione EXG scambia il contenuto di due registri a 32 bit. Lo scambio di una long word può avvenire tra due registri dati, due registri indirizzi o tra un registro dati ed un registro indirizzi.

7-4. Da BCD a binario

Scopo: Convertire quattro cifre BCD contenute nella variabile STRING, alla locazione 6000, in un numero binario e memorizzare il risultato nella variabile NUMBER, alla locazione 6004. La cifra BCD più significativa si trova nella locazione di memoria 6000. C'è una cifra BCD in ciascuno dei byte di STRING.

Problemi Campione:

```
a. Input:   STRING   - (6000) = 02
                       (6001) = 09
                       (6002) = 07
                       (6003) = 01
   Output:  NUMBER   - (6004) = 0B9B16 = 297110
```

```
b. Input:   STRING   - (6000) = 9
                       (6001) = 07
                       (6002) = 00
                       (6003) = 02
   Output:  NUMBER   - (6004) = 25E6₁₆ = 9702₁₀
```

Programma 7-4a:

```
00006000:                 DATA      EQU      $6000
00004000:                 PROGRAM   EQU      $4000
                          ;
00006000:                 STRING    EQU      $6000          IND. STRINGA CON 4 CIFRE BCD
00006004:                 RESULT    EQU      $6004          INDIRIZZO DEL RISULTATO
                          ;
                                    ORG      PROGRAM

00004000: 307C 6000       PGM_7_4A  MOVEA.W  #STRING,A0     PUNTATORE PRIMA CIFRA BCD
00004004: 7003                      MOVEQ    #4-1,D0        NUMERO DI CIFRE (-1)
00004006: 4281                      CLR.L    D1             AZZERA RISULTATO (D1)
00004008: 4282                      CLR.L    D2             AZZERA REG. DELLE CIFRE
0000400A: 6008                      BRA.S    NOMULT         LA PRIMA VOLTA NON MOLTIPLICARE
                          ;
0000400C: D241            LOOP      ADD.W    D1,D1          2X
0000400E: 3601                      MOVE.W   D1,D3
00004010: E54B                      LSL.W    #2,D3          8X = 2X * 4
00004012: D243                      ADD.W    D3,D1          10X = 8X + 2X
                          ;
00004014: 1418            NOMULT    MOVE.B   (A0)+,D2       CIFRA BCD SEG.,(D2[15-8]IMMUTATO)
00004016: D242                      ADD.W    D2,D1          AGGIUNGI CIFRA SUCCESSIVA
00004018: 51C8 FFF2                 DBRA     D0,LOOP        CONTINUA SE CI SONA ANCORA CIFRE
                          ;
0000401C: 31C1 6004                 MOVE.W   D1,RESULT      SALVA IL RISULTATO
                          ;
00004020: 4E75                      RTS

                                    END      PGM_7_4A '
```

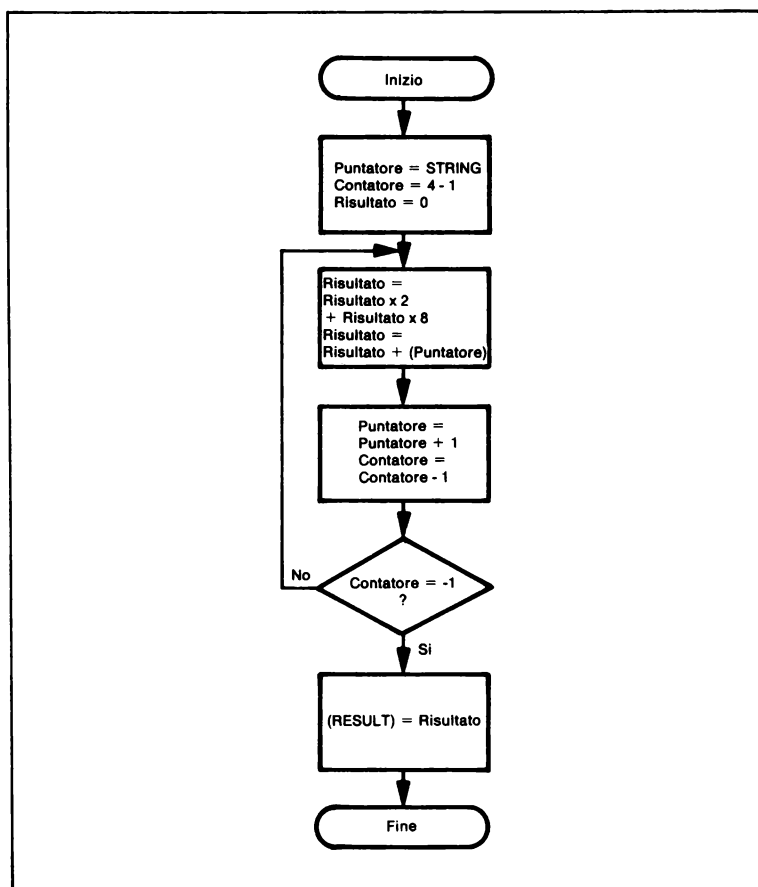
Spiegazione del programma 7-4a

Il programma 7-4a moltiplica ciascun risultato intermedio per 10, usando la formula 10x = 8x + 2x. La moltiplicazione per due richiede uno shift logico a sinistra (LSL), mentre la moltiplicazione per 8 ne richiede tre.

I valori BCD sono convertiti in forma binaria per poter sfruttare gli operatori binari forniti dal processore. Nel caso dell'addizione la rappresentazione binaria richiede meno spazio dell'equivalente formato BCD. Tuttavia, in alcuni casi, il tempo e la memoria necessari per la conversione riducono i vantaggi derivanti dall'uso della forma binaria.

Per sommare la cifra BCD al risultato contenuto nel registro D1 il Programma 7-4a impiega una istruzione ADD.W. Se avessimo usato ADD.B D2,D1 il programma non avrebbe funzionato con tutti i valori. Consideriamo, ad esempio, il valore 0257. Prima di aggiungere la cifra più bassa D1 contiene 0250_{10} o $00FA_{16}$. Sommando 7 al solo byte basso di D1 si ottiene FA + 07 = 01 ed il byte alto rimane ancora uguale a zero. Dato che non possiamo sommare un valore di un byte ad uno di una word, lo abbiamo messo in un registro dati prima di eseguire l'addizione. Perchè non dobbiamo ricorrere ad un'operazione di estensione prima dell'addizione?

La prima moltiplicazione non viene eseguita, poichè sappiamo che il valore iniziale di D2 è 0. Tuttavia, eliminando le istruzioni di salto otterremmo ancora lo stesso risultato.

Diagramma di Flusso 7-4a

Usando il linguaggio assembly vi accorgerete che esistono parecchie soluzioni diverse per uno stesso problema. In questo caso ci siamo serviti dell'istruzione ADD per shiftare a sinistra di una posizione un certo valore, perchè questo è il modo più rapido di effettuare questa operazione sull'MC68000. Due istruzioni ADD sarebbero ancora più rapide dell'istruzione LSL, ma occuperebbero due byte in più.

Potremmo anche ricorrere ad una delle istruzioni di moltiplicazione dell'MC68000 che moltiplicano due operandi a 16 bit, fornendo un risultato a 32 bit in uno dei registri dati. Almeno uno degli operandi deve trovarsi in un registro dati. L'MC68000 consente sia le moltiplicazioni con segno che quelle senza segno. Nel primo caso (MULS) gli operndi sono considerati dei valori provvisti di segno e tale è anche il risultato. Nell'altro caso (MULU) sono tutti valori privi di segno. Nel programma 7-4b abbiamo utilizzato l'istruzione MULU, invece delle istruzioni ADD e LSL del programma precedente.

Programma 7-4b:

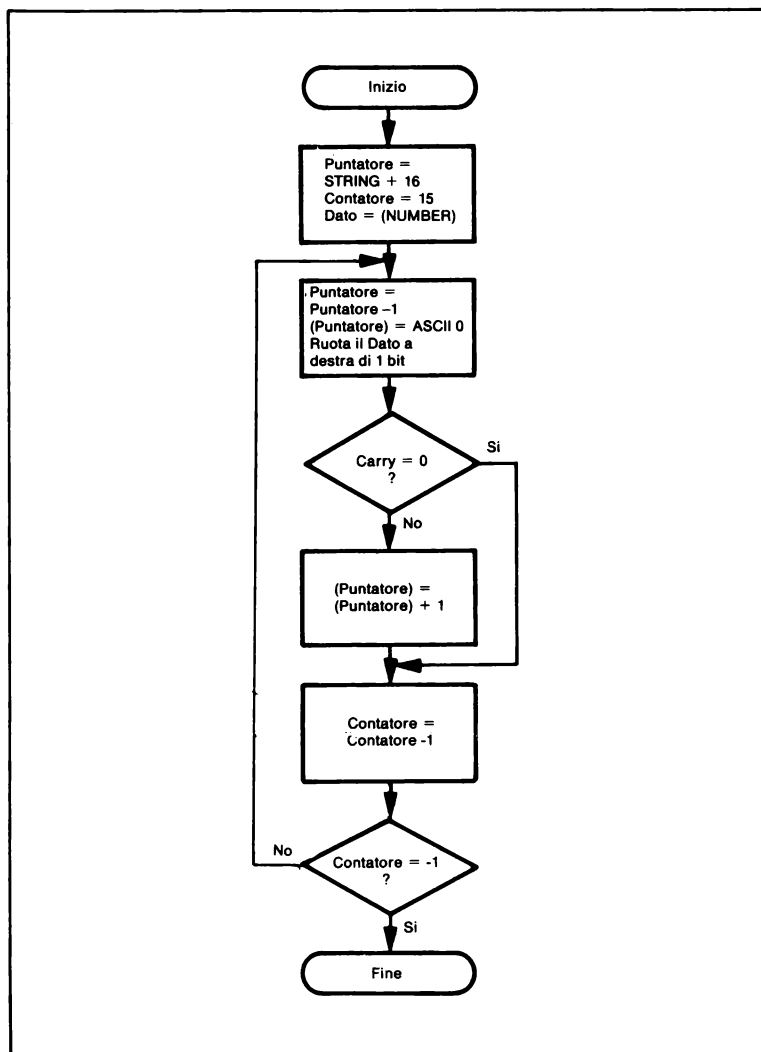
```
00006000:                 DATA      EQU      $6000
00004000:                 PROGRAM   EQU      $4000
                          ;
00006000:                 STRING    EQU      $6000          IND. STRINGA CON 4 CIFRE BCD
00006004:                 CODE      EQU      $6004          INDIRIZZO DEL RISULTATO
                          ;
                                    ORG      PROGRAM

00004000: 307C 6000       PGM_7_4B  MOVEA.W  #STRING,A0     PUNTATORE PRIMA CIFRA BCD
00004004: 7003                      MOVEQ    #4-1,D0        NUMERO DI CIFRE (-1)
00004006: 4281                      CLR.L    D1             AZZERA RISULTATO (D1)
00004008: 4282                      CLR.L    D2             AZZERA REG. DELLE CIFRE
0000400A: 6004                      BRA.S    NOMULT         LA PRIMA VOLTA NON MOLTIPLICARE
                          ;
0000400C: C2FC 000A       LOOP      MULU     #10,D1         D1 = D1 * 10
                          ;
00004010: 1418            NOMULT    MOVE.B   (A0)+,D2       CIFRA BCD SEG.,(D2[15-8]IMMUTATO)
00004012: D242                      ADD.W    D2,D1          AGGIUNGI CIFRA SUCCESSIVA
00004014: 51C8 FFF6                 DBRA     D0,LOOP        CONTINUA SE CI SONA ANCORA CIFRE
                          ;
00004018: 31C1 6004                 MOVE.W   D1,CODE        SALVA IL RISULTATO
                          ;
0000401C: 4E75                      RTS

                                    END      PGM_7_4B
```

7-5. Conversione di un numero binario in una stringa ASCII

Scopo: Convertire il numero binario contenuto nella variabile NUMBER, alla locazione 6000, in 16 caratteri ASCII (0 oppure 1) che saranno memorizzati nella variabile stringa STRING, posta alla locazione di memoria 6002.

Problema Campione:

```
Input:   NUMBER   - (6000) = 31D2 = 0011 0001 1101 0010
Output:  STRING   - (6002) = 30 '0'
                    (6003) =  0 '0'
                    (6004) = 31 '1'
                    (6005) = 31 '1'
                    (6006) = 30 '0'
                    (6007) = 30 '0'
                    (6008) = 30 '0'
                    (6009) = 31 '1'
                    (600A) = 31 '1'
                    (600B) = 31 '1'
                    (600C) = 30 '0'
                    (600D) = 31 '1'
                    (600E) = 30 '0'
                    (600F) = 30 '0'
                    (6010) = 31 '1'
                    (6011) = 30 '0'
```

Diagramma di Flusso 7-5

Spiegazione del programma 7-5

Le cifre ASCII formano una sequenza tale che ASCII 1 = ASCII 0 + 1. L'istruzione ADD può essere usata per incrementare direttamente il contenuto di una locazione di memoria. Non è necessaria, quindi, nessuna istruzione esplicita per caricare un dato dalla memoria in un registro, incrementarlo e salvare, successivamente, il risultato in memoria. Nessun registro viene modificato.

Il puntatore (A0) della stringa indica, inizialmente, la fine della stringa +1 ($6002+16_{10}$) e viene decrementato all'inizio di ciascuna iterazione. Accedendo ai dati in questo modo l'indirizzo di fine stringa è, in realtà, l'indirizzo del primo byte fuori dalla stringa. Ad esempio, il byte a $6002 + 16_{10}$ non si trova nella stringa di cifre

Programma 7-5

```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
00006000:                NUMBER   EQU      $6000              IND. DEL NUMERO A 16 BIT
00006002:                STRING   EQU      $6002              IND. DELLA STRINGA ASCII EQUIV.
                         ;
                                  ORG      PROGRAM

00004000: 207C 0000
00004004: 6012           PGM_7_5  MOVEA.L  #STRING+16,A0      PUNTATORE FINE STRINGA (+1)
00004006: 700F                    MOVEQ    #15,D0             CONTATORE DEL CICLO (-1)
00004008: 123C 0030               MOVE.B   #'0',D1
0000400C: 3438 6000               MOVE.W   NUMBER,D2          PRENDI IL DATO NUMERICO
                         ;
00004010: 1101           LOOP     MOVE.B   D1,-(A0)           SI PRESUME CHE LSB = 0
00004012: E25A                    ROR.W    #1,D2              CONTROLLA LSB
00004014: 6404                    BCC.S    LOOPEND            SE E' 0 PROVA COL SUCCESSIVO BIT
                         ;
00004016: 0610 0001               ADDI.B   #1,(A0)            CAMBIA '0' IN '1'
                         ;
0000401A: 51C8 FFF4      LOOPEND  DBRA     D0,LOOP            ESAMINA TUTTI I BIT
                         ;
0000401E: 4E75                    RTS

                                  END      PGM_7_5
```

ASCII. Infine, notate che 6002 + 16_{10} si identifica più facilmente con una stringa a 16 bit, invece di 6002 + 15_{10}.

Utilità della conversione binario - ASCII

La conversione binario-ASCII è necessaria quando devono essere stampati dei numeri in forma binaria su una periferica ASCII. Degli output binari sono utili nella fase di debugging e di collaudo, quando ogni bit ha un significato particolare; esempi tipici sono gli input provenienti da un gruppo di switch sul pannello frontale o gli output destinati ad un insieme di LED. Se un programmatore non dispone di questi valori in forma binaria, per poter controllare i singoli bit deve effettuare una conversione manuale, soggetta a possibili errori.

PROBLEMI

7-1. Da ASCII ad esadecimale

Scopo: Convertire il contenuto della variabile A-DIGIT, alla locazione di memoria 6000, da carattere ASCII a cifra esadecimale e salvare il risultato nella variabile H-DIGIT, alla locazione di memoria 6001. Partire dal presupposto che A-DIGIT contenga la rappresentazione ASCII di una cifra esadecimale (7 bit con MSB = 0).

Problemi Campione:

a. Input:	A-DIGIT	- (6000)	=	43 'C'
Output:	H-DIGIT	- (6001)	=	0C
b. Input:	A-DIGIT	- (6000)	=	36 '6'
Output:	H-DIGIT	- (6001)	=	06

7-2. Da sette segmenti a decimale

Scopo: Convertire il contenuto della variabile CODE, alla locazione di memoria 6000, da un codice a sette segmenti ad un numero decimale e salvare il risultato nella variabile NUMBER, alla locazione di memoria 6001. Se CODE non contiene un codice a sette segmenti valido, porre NUMBER a FF_{16}. Usare la tabella a sette segmenti della Figura 7-1 e provare a confrontare i codici.

Problemi Campione:

a. Input:	CODE	- (6000)	=	4F
Output:	NUMBER	- (6001)	=	03
b. Input:	CODE	- (6000)	=	28
Output:	NUMBER	- (6001)	=	FF

7-3. Da decimale ad ASCII

Scopo: Convertire il contenuto della variabile DIGIT, alla locazione di memoria 6000, da cifra decimale a carattere ASCII e salvare il risultato nella variabile CHAR, alla locazione di memoria 6001. Se il numero in DIGIT non è una cifra decimale porre in CHAR il carattere ASCII di spazio (20_{16}).

Problemi Campione:

a. Input:	DIGIT	- (6000)	=	07
Output:	CHAR	- (6001)	=	37 '7'

b. Input: DIGIT – (6000) = 55
Output: CHAR – (6001) = 20 spazio

7-4. Da binario a BCD

Scopo: Convertire il contenuto della variabile NUMBER, alla locazione di memoria 6000, in quattro cifre BCD nella variabile STRING, alla locazione di memoria 6002 (la cifra più significativa in 6002). Il numero a 16 bit in NUMBER è privo di segno e minore di 10.000.

Problema Campione:

```
Input:   NUMBER   – (6000) = 1C52 (7250 decimale)
Output:  STRING   – (6002) = 07
                    (6003) = 02
                    (6004) = 05
                    (6005) = 00
```

7-5. Da stringa ASCII a numero binario

Scopo: Convertire gli otto caratteri ASCII nella variabile STRING, che inizia alla locazione 6000, in un numero binario ad 8 bit nella variabile NUMBER, alla locazione 6008 (il carattere più significativo è nella locazione 6000). Se sono tutti caratteri ASCII 1 oppure ASCII 0 azzerare la variabile di un byte ERROR, alla locazione 6009, altrimenti mettere FF_{16} sempre in ERROR.

Problemi Campione:

```
a. Input:   STRING   – (6000) = 31 '1'
                       (6001) = 31 '1'
                       (6002) = 30 '0'
                       (6003) = 31 '1'
                       (6004) = 30 '0'
                       (6005) = 30 '0'
                       (6006) = 31 '1'
                       (6007) = 30 '0'

   Output:  NUMBER   – (6008) = D2
                       (6009) = 0
```

b. Input: Come nell'esempio precedente tranne
(6005) = 37 '7'

Output: ERROR - (6009) = FF

BIBLIOGRAFIA

Altri metodi di conversione da BCD a binario sono trattati in M.L. Roginsky e J.A. Tabb, "Microprocessor Algorithms Make BCD-Binary Conversons Super-fast", EDN, 5 Gennaio 1977, pagg.46-50; e in J.B. Peatman, *Microcomputer-based Design,* Mac Graw-Hill, New York, 1977, pagg.400-406.

CAPITOLO 8

PROBLEMI ARITMETICI

OPERAZIONI ARITMETICHE DECIMALI E CON WORD MULTIPLE

Rappresentazione dei numeri con segno

Nelle applicazioni dei microprocessori, gran parte delle operazioni aritmetiche consistono in manipolazioni binarie o decimali di word multiple. Molto spesso una correzione decimale (decimal adjust) o poche altre operazioni di questo tipo sono le sole istruzioni aritmetiche disponibili oltre a quelle di addizione e sottrazione. In questi casi si possono eseguire le varie operazioni aritmetiche ricorrendo a delle sequenze di istruzioni. L'MC68000, tuttavia, dispone di istruzioni di moltiplicazione e divisione, con o senza segno, per i numeri binari a 16 bit oltre, naturalmente, alle istruzioni di addizione e sottrazione.

L'MC68000 consente di eseguire operazioni aritmetiche su numeri binari provvisti o meno di segno. I numeri con segno sono rappresentati in complemento a due: questo significa che le operazioni di addizione e sottrazione sono identiche indipendentemente dalla presenza o meno del segno, al contrario di quanto accade per la moltiplicazione e la divisione, che necessitano di istruzioni differenti. Provate alcuni degli esempi per convincervi.

Esecuzione delle operazioni in precisione multipla

L'aritmetica binaria in precisione multipla comporta semplicemente la ripetizione di determinate istruzioni fondamentali. Ci serviamo del bit di Extend per trasferire informazioni fra word differenti: è messo a 1 quando un'addizione genera un riporto o una sottrazione dà origine ad un prestito. 'Add with Extend' e 'Subtract with Extend' utilizzano questa informazione prodotta dall'operazione precedente. Prima di iniziare dovete ricordarvi di azzerare il bit di Extend. (Evidentemente non esiste un riporto o un prestito quando si tratta dei bit meno significativi).

L'aritmetica decimale è una funzione abbastanza comune per i microprocessori, i quali, normalmente, dispongono di istruzioni speciali destinate a questo scopo, che eseguono direttamente le operazioni decimali oppure correggono i risultati delle operazioni binarie nella forma decimale appropriata. L'aritmetica decimale è essenziale in applicazioni quali terminali di punti vendita, processori di controllo, sistemi d'inserimento ordini e terminali bancari. L'MC68000 fornisce istruzioni per l'addizione e la sottrazione decimali e, quindi, dato che esegue direttamente le operazioni aritmetiche decimali, non ha bisogno di istruzioni di correzione, come accade per altri microprocessori.

È possibile eseguire la moltilicazione e la divisione decimali, mediante una serie, rispettivamente, di addizioni e sottrazioni. Nel primo caso, i risultati occupano uno spazio maggiore, poichè una moltiplicazione genera un risultato due volte più lungo degli operandi. La divisione, al contrario, produce un risultato di lunghezza inferiore. Le moltiplicazioni e le divisioni richiedono molto tempo quando sono eseguite tramite software, a causa delle ripetute operazioni necessarie.

ESEMPI DI PROGRAMMAZIONE

8-1. Addizione binaria a 64 bit

Scopo: Sommare due numeri binari di quattro word (64 bit). Il primo numero è la variabile a 64 bit NUM1 e occupa le locazioni di memoria comprese fra 6000 e 6007, il secondo è la variabile a 64 bit NUM2 e occupa le locazioni da 6200 a 6207. Mettere la somma in NUM1, alle locazioni 6000-6007.

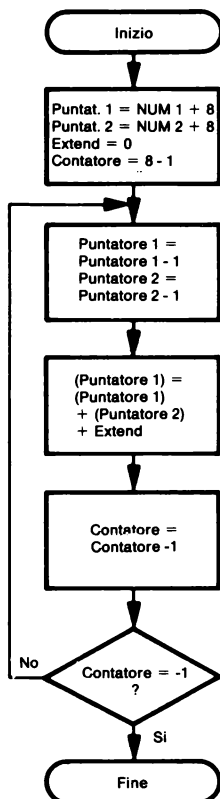
Problema Campione:

```
Input:   NUM1   - (6000) = 6A4D
                  (6002) = ED05  6A4DED05A9376414₁₆
                  (6004) = A937  è il primo numero
                  (6006) = 6414
         NUM2   - (6200) = 56C8
                  (6202) = 46E6  56C846E676C84AEA₁₆
                  (6204) = 76C8  è il secondo numero
                  (6206) = 4AEA

Output:  NUM1   - (6000) = C116
                  (6002) = 33EC  C11633EC1FFFAEFE₁₆
                  (6004) = 1FFF  è la somma
                  (6006) = AEFE
```

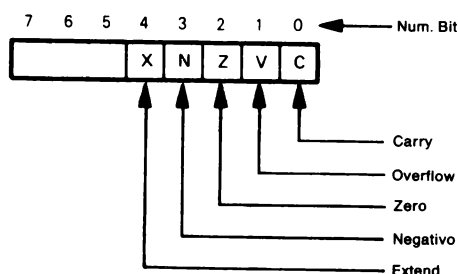
Programma 8-1a:

```
00006000:                 DATA       EQU     $6000
00004000:                 PROGRAM    EQU     $4000
                          ;
00006000:                 NUM1       EQU     $6000             IND. DEL PRIMO NUM. BINARIO A 64 BIT
00006200:                 NUM2       EQU     $6200             IND. DEL SEC. NUM. BINARIO A 64 BIT
00000008:                 BYTECOUNT  EQU     $8                NUMERO DI BYTE DA SOMMARE
                          ;
                                     ORG     PROGRAM

00004000: 207C 0000
00004004: 6000            PGM_8_1A MOVEA.L #NUM1+BYTECOUNT,A0 IND. OLTRE LA FINE DEL PRIMO NUM.
00004006: 227C 0000
0000400A: 6208                     MOVEA.L #NUM2+BYTECOUNT,A1 IND. OLTRE LA FINE DEL SEC. NUM.
0000400C: 44FC 0000                MOVE    #0,CCR             AZZERA FLAG EXTEND (E GLI ALTRI)
00004010: 7407                     MOVEQ   #BYTECOUNT-1,D2 CONT. LOOP CORRETTO PER DBRA
```

```
                               ;
00004012: 1020          LOOP    MOVE.B  -(A0),D0
00004014: 1221                  MOVE.B  -(A1),D1
00004016: D101                  ADDX.B  D1,D0       D0[0-7]:=D0[0-7] + D1[0-7] + (EXT)
00004018: 1080                  MOVE.B  D0,(A0)     SALVA RISULTATO
0000401A: 51CA FFF6             DBRA    D2,LOOP     CONTINUA
                        ;
0000401E: 4E75                  RTS

                                END     PGM_8_1A
```

Diagramma di Flusso 8-1

Mettere i Flag a Zero o a Uno

L'istruzione MOVE TO CCR azzera o mette a uno tutti *i codici di condizione* nel registro di stato del processore, in base al contenuto dell'operando sorgente. Benchè quest'ultimo abbia sempre la lunghezza di una word (16 bit) viene impiegato solo il byte meno significativo per determinare i codici di condizione. Dunque MOVE #0,CCR azzera tutti i flag di stato (Negativo, Zero, Overflow, Carry ed Extend). Questa istruzione serve ad azzerare il flag di Extend, in previsione della prima istruzione ADDX.

MOVE TO CCR non è la sola istruzione in grado esplicitamente di modificare il contenuto del registro di stato. Possiamo servirci anche delle istruzioni immediate ANDI, EORI ed ORI per azzerare, complementare e porre a 1, selettivamente, i singoli flag. Ad esempio, mediante l'istruzione ANDI #$EF,CCR azzeriamo soltanto il flag di Extend, senza modificare gli altri codici di condizione. Il formato dell'operando immediato è il seguente:

Add con Extend

L'istruzione ADDX (Add with Extend) somma il contenuto di due registri. Se il flag di Extend vale uno, allora al risultato viene aggiunto 1. Oltre ad eseguire l'addizione, ADDX modifica, in modo appropriato, il flag di Extend per le operazioni successive. Osservate come, in questo ciclo di programma, nessun'altra istruzione alteri lo stato del flag di Extend.

Il flag di Extend

Il flag di Extend è simile al flag di Carry della maggior parte degli altri microprocessori. L'MC68000 li ha entrambi. Come regola generale, il flag di Carry è messo a 1 se si produce un riporto nel bit più significativo del risultato di un'addizione oppure se si verifica un prestito in una sottrazione; altrimenti viene azzerato. Il flag di Extend rispecchia, generalmente , lo stato del flag di Carry, tranne che durante il trasferimento di dati, quando il suo valore non viene modificato.

Somma di Operandi in Memoria

Spiegazione del programma 8-1b

Un metodo di addizione più rapido ed elegante viene mostrato nel Programma 8-1b, che usa la seconda forma dell'istruzione 'Add with Extend': quella, ben più potente, da memoria a memoria di cui dispone l'MC68000. Questo formato richiede l'uso di due registri indirizzi che puntano agli operandi in memoria. I registri indirizzi sono *decrementati* in base alla lunghezza degli operandi, prima di essere utilizzati per prelevare questi ultimi. L'istruzione 'Add with Extend' è in grado di operare su dati da 8, 16 o 32 bit.

Programma 8-1b:

```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
00006000:                NUM1     EQU      $6000          IND. DEL PRIMO NUM. BINARIO A 64 BIT
00006200:                NUM2     EQU      $6200          IND. DEL SEC. NUM. BINARIO A 64 BIT
                         ;
                                  ORG      PROGRAM

00004000: 207C 0000
00004004: 6008           PGM_8_1B MOVEA.L  #NUM1+8,A0     IND. OLTRE LA FINE DEL PRIMO NUM.
00004006: 227C 0000
0000400A: 6208                    MOVEA.L  #NUM2+8,A1     IND. OLTRE LA FINE DEL SEC. NUM.
0000400C: 44FC 0000               MOVE     #0,CCR         AZZERA FLAG EXTEND (E GLI ALTRI)
                         ;
00004010: D189                    ADDX.L   -(A1),-(A0)    SOMMA LE LONG WORD DI ORDINE BASSO
                         ;                                ... E METTI RISULTATO IN NUM1
00004012: D189                    ADDX.L   -(A1),-(A0)    SOMMA LE LONG WORD DI ORDINE ALTO
                         ;                                ... E METTI IL RISULTATO IN NUM1
00004014: 4E75                    RTS

                                  END      PGM_8_1B
```

L'istruzione di somma binaria ADD

L'MC68000 prevede anche la possibilità di eseguire un'addizione binaria mediante l'istruzione ADD, del tutto simile a quella ADDX, tranne per il fatto di non usare lo stato del flag di Extend. L'istruzione ADD richiede anche che almeno uno degli operandi si trovi in un registro dati. Come si potrebbe modificare il Programma 8-1a in modo da utilizzare l'istruzione ADD, anzichè quella ADDX?

Precisione Decimale in Rappresentazione Binaria

Calcolo del numero di bit necessari per rappresentare un dato numero di cifre decimali

Salvare dei dati in formato binario, anzichè decimale, richiede un minore impiego di memoria. Ad esempio, **dieci bit corrispondono, approssimativamente, a tre cifre decimali,** poichè $2^{10} = 1024$. **In questo modo è possibile calcolare, con la formula seguente, il numero approssimativo di bit necessari per ottenere una determinata precisione in cifre decimali:**

Numero di bit = (10/3) x Numero di cifre decimali

Così, una precisione di dodici cifre richiederà:

$$12 \times 10/3 = 40 \text{ bit}$$

8-2. Addizione binaria a 64 bit

Scopo: Sommare due numeri BCD di lunghezza superiore ad un byte. La lunghezza in byte è definita dalla variabile LENGTH, alla locazione 6000. Il primo numero (i bit più significativi per primi) è contenuto nella variabile BCDNUM1, alla locazione 6001. Il secondo numero è contenuto nella variabile BCDNUM2, alla locazione 6101. La somma sostituisce il numero in BCDNUM1. Ciascuno dei byte dei numeri BCD contiene due cifre decimali.

Problema Campione:

```
Input:   LENGTH    - (6000) = 04 Byte in ogni numero
         BCDNUM1   - (6001) = 36
                     (6002) = 70 36701985 Primo numero
                     (6003) = 19
                     (6004) = 85

         BCDNUM2   - (6101) = 12
                     (6102) = 66 12663459 Secondo numero
                     (6103) = 34
                     (6104) = 59

Output:  BCDNUM1   - (6001) = 49
                     (6002) = 36 49365444 è la somma
                                 decimale
                     (6003) = 54
                     (6004) = 44

Cioè:     36701985
         +12663459
         ---------
          49365444
```

Programma 8-2a:

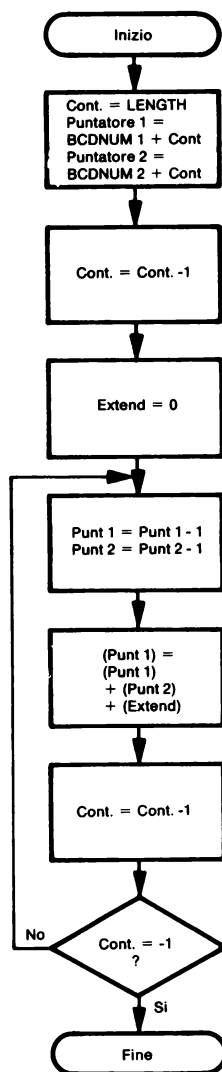
```
00006000:                DATA     EQU     $6000
00004000:                PROGRAM  EQU     $4000
                         ;
00006000:                LENGTH   EQU     $6000            LUNGH.IN BYTE DEL NUMERO BCD
00006001:                BCDNUM1  EQU     $6001            IND. DEL PRIMO NUMERO BCD
00006101:                BCDNUM2  EQU     $6101            IND. DEL SECONDO NUMERO BCD
                         ;
                                  ORG     PROGRAM

00004000: 4242           PGM_8_2A CLR.W   D2
00004002: 1438 6000               MOVE.B  LENGTH,D2
00004006: 3442                    MOVE.W  D2,A2            A2[0-31] = BYTE DEL NUMERO BCD
00004008: 41EA 6001               LEA     BCDNUM1(A2),A0   PUNTA OLTRE LA FINE DI BCDNUM1
0000400C: 43EA 6101               LEA     BCDNUM2(A2),A1   PUNTA OLTRE LA FINE DI BCDNUM2
                         ;
00004010: 5342                    SUBQ    #1,D2            CORREGGI LUNGH. PER FINE LOOP
00004012: 44FC 0000               MOVE    #0,CCR           AZZERA FLAG EXTEND PER ABCD
                         ;
00004016: C109           LOOP     ABCD.B  -(A1),-(A0)      ADDIZIONE BCD CON EXTEND
00004018: 51CA FFFC               DBRA    D2,LOOP          CONTINUA
                         ;
0000401C: 4E75                    RTS
                                  END     PGM_8_2A A
```

L'istruzione ABCD

L'MC6800, a differenza della maggior parte dei microprocessori, esegue l'addizione decimale con una singola istruzione ABCD (Add Decimal with Extend). Al pari dell'istruzione ADDX, ABCD effettua l'addizione usando il valore del flag di Extend. Tuttavia, dal momento che viene impiegata l'aritmetica BCD, non è necessaria un'istruzione di correzione decimale, del tipo di quella DAA del microprocessore Motorola 6809. L'MC68000 dispone anche di un'istruzione per la sottrazione decimale (SBCD).

Spiegazione del programma 8-2a

Il Programma 8-2a utilizza l'istruzione LEA (Load Effective Address) per calcolare l'indirizzo dell'ultimo byte del numero decimale più uno. Essa calcola l'indirizzo effettivo nel modo consueto, ma, poi, lo mette semplicemente nel registro indirizzi specificato, anzichè usarlo per trasferire dei dati, rendendolo disponibile per una successiva utilizzazione, senza che debba essere ricalcolato.

Diagramma di Flusso 8-2

Limitazione del programma 8-2a

Bisogna notare che l'impiego dell'indirizzamento indiretto a registro con spostamento, insieme all'istruzione LEA, comporta alcune restrizioni nel Programma 8-2a: dal momento che lo spostamento (BCDNUM1), facendo parte dell'operando, sarà al massimo di 16 bit, non può essere utilizzata tutta la memoria che il microprocessore è capace di indirizzare. Possiamo rendere il Programma 8-2a adatto ad un impiego più generico, permettendogli di utilizzare l'intero spazio indirizzabile, anche se questo finisce per rendere necessarie parecchie istruzioni addizionali. Il Programma 8-2b illustra questo tipo di soluzione.

Programma 8-2b:

```
00006000:                  DATA     EQU      $6000
00004000:                  PROGRAM  EQU      $4000
                           ;
00006000:                  LENGTH   EQU      $6000          LUNGH.IN BYTE DEL NUMERO BCD
00006001:                  BCDNUM1  EQU      $6001          IND. DEL PRIMO NUMERO BCD
00006101:                  BCDNUM2  EQU      $6101          IND. DEL SECONDO NUMERO BCD
                           ;
                                    ORG      PROGRAM

00004000: 4242             PGM_8_2B CLR      D2
00004002: 1438 6000                 MOVE.B   LENGTH,D2
00004006: 207C 0000
0000400A: 6001                      MOVEA.L  #BCDNUM1,A0    PUNTAT. INIZIO DI BCDNUM1
0000400C: 227C 0000
00004010: 6101                      MOVEA.L  #BCDNUM2,A1    PUNTAT. INIZIO DI BCDNUM2
00004012: 41F0 2000                 LEA      0(A0,D2.W),A0  PUNTA OLTRE LA FINE DEL VALORE 1
00004016: 43F1 2000                 LEA      0(A1,D2.W),A1  PUNTA OLTRE LA FINE DEL VALORE 2
                           ;
0000401A: 5342                      SUBQ.W   #1,D2          CORREGGI LUNGH. PER FINE LOOP
0000401C: 44FC 0000                 MOVE     #0,CCR         AZZERA FLAG EXTEND PER ABCD
                           ;
00004020: C109             LOOP     ABCD.B   -(A1),-(A0)    ADDIZIONE BCD CON EXTEND
00004022: 51CA FFFC                 DBRA     D2,LOOP        CONTINUA
                           ;
00004026: 4E75                      RTS

                                    END      PGM_8_2B
```

La procedura usata in entrambi questi programmi è capace di sommare numeri decimali (BCD) di qualsiasi grandezza (fino a 131.072 cifre!). Dato che ogni cifra decimale richiede quattro bit, una precisione di dodici cifre richiederà:

12 x 4 = 48 bit

rispetto ai 40 bit di un'addizione binaria. Sono necessari sei byte anzichè cinque: un incremento del 20%.

Sostituendo l'istruzione ABCD dei Programmi 8-2a e 8-2b con ADDX otterremmo una soluzione al problema dell'addizione binaria più generale, rispetto a quella fornita dal Programma 8-1.

8-3. Moltiplicazione binaria a 16 bit

Scopo: Moltiplicare il numero a 16 bit privo di segno della variabile NUM1, alla locazione 6000, per il numero binario a 16 bit privo di segno della variabile NUM2, alla locazione 6002. Mettere il risultato a 32 bit nella variabile long word RESULT, alla locazione 6004, con i 16 bit più significativi nella locazione 6004 ed i 16 meno significativi nella locazione 6006.

Problemi Campione:

```
a. Input:   NUM1     - (6000) = 0003
            NUM2     - (6002) = 0005

   Output:  RESULT   - (6004) = 0000
                       (6006) = 000F
                       o in decimale 3 x 5 = 15
```

```
b. Input:   NUM1      - (6000) = 706F
            NUM2      - (6002) = 0161

   Output:  RESULT    - (6004) = 009B
                        (6006) = 090F
                        o in decimale 28783 x 353 = 10160399
```

Programma 8-3a:

```
00006000:                DATA     EQU     $6000
00004000:                PROGRAM  EQU     $4000
                         ;
                                  ORG     DATA

00006000:                NUM1     DS.W    1            MOLTIPLICANDO A 16 BIT
00006002:                NUM2     DS.W    1            MOLTIPLICATORE  A 16 BIT
00006004:                RESULT   DS.L     1            RISULTATO MOLTIPLICAZIONE A 32 BIT
                         ;
                                  ORG     PROGRAM

00004000: 3038 6000      PGM_8_3A MOVE.W  NUM1,D0      MOLTIPLICANDO
00004004: C0F8 6002               MULU    NUM2,D0      MOLTIPLICAZIONE SENZA SEGNO
00004008: 21C0 6004               MOVE.L  D0,RESULT     SALVA RISULTATO MOLTIPL. A 32 BIT
                         ;
0000400C: 4E75                    RTS

                                  END     PGM_8_3A
```

L'MC68000 esegue moltiplicazioni e divisioni binarie su numeri con e senza segno. Per moltiplicare due numeri binari a 16 bit con segno è sufficiente sostituire l'istruzione MULU con quella MULS (Multiply Signed).

Oltre alle utilizzazioni più ovvie, come ad esempio nei terminali dei punti vendita, la moltiplicazione è anche una parte essenziale di molti algoritmi matematici. La velocità con cui un processore esegue una moltiplicazione determina la sua utilità nei controlli di processo, nel controllo adattivo, nella rilevazione e l'analisi di un segnale.

Matrici Multidimensionali

La moltiplicazione viene usata comunemente anche nella localizzazione di elementi appartenenti a matrici multidimensionali. Ad esempio, disponendo di una matrice di valori letti da sensori, organizzati sulla base del numero della stazione di rilevamento e del numero del sensore, possiamo indicare il valore letto dal settimo sensore della stazione numero 5 con R (5,7), dove R è il nome della matrice. Il metodo comunemente adottato per salvare in memoria una matrice di questo tipo è quello di iniziare con R (0,0) all'indirizzo RBASE e proseguire con R (0,1), ecc. Se ci sono tre stazioni (0,1 e 2) e quattro sensori per ogni stazione (0, 1, 2 e 3) metteremo le rilevazioni nelle seguenti locazioni di memoria:

Locazioni di Memoria	**Valori**
RBASE	R(0,0)
RBASE + 1	R(0,1)
RBASE + 2	R(0,2)
RBASE + 3	R(0,3)
RBASE + 4	R(1,0)

RBASE + 5	R(1,1)
RBASE + 6	R(1,2)
RBASE + 7	R(1,3)
RBASE + 8	R(2,0)
RBASE + 9	R(2,1)
RBASE + 10	R(2,2)
RBASE + 11	R(2,3)

In generale, se noi conosciamo il numero della stazione I ed il numero del sensore J, il valore R (I,J) si troverà all'indirizzo:

RBASE + (N x I) + J

dove N è il numero dei sensori in ciascuna stazione. Perciò, la localizzazione di un determinato valore allo scopo di aggiornarlo, visualizzarlo su un terminale oppure utilizzarlo per una qualche operazione matematica, richiede una moltiplicazione. Un operatore, ad esempio, può chiedere ad uno strumento la stampa del valore letto in quel momento dal sensore 03 della stazione 02. Per trovare quel valore, bisogna calcolarne l'indirizzo

RBASE + (4 x 2) + 3 = RBASE + 11

Per una matrice con più di due dimensioni sarebbe necessario un numero maggiore di moltiplicazioni. Ad esempio, potremmo organizzare i sensori in base al numero della stazione, alla posizione nella direzione X e nella direzione Y. (Ogni stazione avrebbe, così, i sensori disposti ad intervalli regolari su una superficie bidimensionale). In questo modo, sarebbe possibile individuare un valore R (2,3,1), rilevato del sensore della stazione 02, che occupa la posizione 03 sull'asse X e 01 sull'asse Y. L'aggiunta di altre dimensioni, come la posizione verticale, il tipo di sensore o l'orario di lettura, costringono il processore ad un numero ancora maggiore di moltiplicazioni, dovendo sempre localizzare gli elementi in una memoria sostanzialmente unidimensionale.

Un Algoritmo di Moltiplicazione Binaria

Esecuzione di una moltiplicazione senza utilizzare le istruzioni MUL

È interessante analizzare una routine di moltiplicazione binaria per due motivi: innanzitutto possiamo confrontarne il tempo di esecuzione con le istruzioni MULU e MULS; e, in secondo luogo, alcuni microprocessori non hanno istruzioni di moltiplicazione, per cui è importante riuscire a capire come funziona una routine di questo tipo.

Si può eseguire una moltiplicazione su un computer nello stesso modo in cui si esegue manualmente una lunga moltiplicazione. Dal momento che si tratta di numeri binari sarà necessario soltanto moltiplicare per 0 o per 1: una moltiplicazione per zero dà ovviamente come risultato zero; mentre moltiplicando per uno

si ottiene lo stesso numero di partenza (il moltiplicando). Così, ogni fase di una moltiplicazione binaria può essere ricondotta alla seguente operazione: se un bit del moltiplicatore è 1 sommare il moltiplicando al prodotto parziale.

L'unico problema che rimane è di accertarsi, ogni volta, di effettuare correttamente gli incolonnamenti. Le operazioni seguenti hanno questa funzione.

1. Shiftare il moltiplicatore di un bit verso sinistra, in modo che il bit da esaminare venga a trovarsi nel Carry.
2. Shiftare il prodotto di un bit verso sinistra, in modo che la successiva addizione sia incolonnata correttamente.

Per rendere le cose più semplici, moltiplicheremo due valori ad 8 bit ottenendo un risultato a 16 bit.

Fase 1 - Inizializzazione

Prodotto = 0
Contatore = 8

Fase 2 - Shiftare il Prodotto per incolonnarlo correttamente

Prodotto = 2 x Prodotto (LSB = 0)

Fase 3 - Shiftare il Moltiplicatore in modo che il bit finisca nel Carry

Moltiplicatore = 2 x Moltiplicatore

Fase 4 - Se il Carry è 1, sommare il moltiplicando al Prodotto

Se Carry = 1, Prodotto = Prodotto + Moltiplicando

Fase 5 - Decrementare il Contatore e controllare se il suo valore è zero

Contatore = Contatore - 1
Se Contatore > 0 vai a Fase 2

Se il moltiplicatore fosse 61_{16} e il moltiplicando $6F_{16}$, l'algoritmo funzionerebbe nel modo seguente.

Inizializzazione:

Prodotto	0000	=	0000000000000000_2
Moltiplicatore	61	=	01100001_2
Moltiplicando	6F	=	01101111_2
Contatore	08		

Dopo la prima iterazione del passo 2-5:

Prodotto	0000	=	0000000000000000_2
Moltiplicatore	C2	=	11000010_2
Moltiplicando	6F	=	01101111_2
Contatore	07		
Carry dal Moltiplicatore	0		

Dopo la seconda iterazione:

Prodotto	006F	=	0000000001101111_2
Moltiplicatore	84	=	10000100_2
Moltiplicando	6F	=	01101111_2
Contatore	06		
Carry dal Moltiplicatore	1		

Dopo la terza iterazione:

Prodotto	014D	=	0000000101001101_2
Moltiplicatore	08	=	00001000_2
Moltiplicando	6F	=	01101111_2
Contatore	05		
Carry dal Moltiplicatore	1		

Dopo la quarta iterazione:

Prodotto	029A	=	0000001010011010_2
Moltiplicatore	10	=	00010000_2
Moltiplicando	6F	=	01101111_2
Contatore	04		
Carry dal Moltiplicatore	0		

Dopo la quinta iterazione:

Prodotto	0534	=	0000010100110100_2
Moltiplicatore	20	=	00100000_2
Moltiplicando	6F	=	01101111_2
Contatore	03		
Carry dal Moltiplicatore	0		

Dopo la sesta iterazione:

Prodotto	0A68	=	0000101001101000_2
Moltiplicatore	40	=	01000000_2
Moltiplicando	6F	=	01101111_2
Contatore	02		
Carry dal Moltiplicatore	0		

Dopo la settima iterazione:

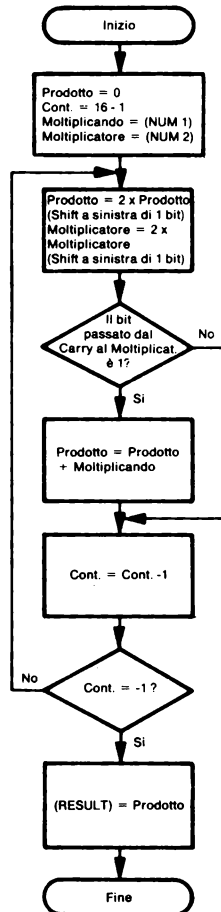
Prodotto	14D0	=	0001010011010000_2
Moltiplicatore	80	=	10000000_2
Moltiplicando	6F	=	01101111_2
Contatore	01		
Carry dal Moltiplicatore	0		

Dopo l'ottava iterazione:

Prodotto	2A0F	=	0010101000001111_2
Moltiplicatore	00	=	00000000_2
Moltiplicando	6F	=	01101111_2
Contatore	00		
Carry dal Moltiplicatore	1		

Programma 8-3b

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
                                     ORG      DATA

00006000:                  NUM1      DS.W     1            MOLTIPLICANDO A 16 BIT
00006002:                  NUM2      DS.W     1            MOLTIPLICATORE  A 16 BIT
00006004:                  RESULT    DS.L     1            RISULTATO MOLTIPLICAZIONE A 32 BIT
                           ;
                                     ORG      PROGRAM

00004000: 4280             PGM_8_3B  CLR.L    D0           AZZERA PRODOTTO A 32 BIT
00004002: 2200                       MOVE.L   D0,D1        LA WORD PIU' ALTA DEVE ESSERE
                           ;                               AZZERATA PER ADD.L
00004004: 3238 6000                  MOVE.W   NUM1,D1      MOLTIPLICANDO A 16 BIT
00004008: 3438 6002                  MOVE.W   NUM2,D2      MOLTIPLICATORE A 16 BIT
0000400C: 760F                       MOVEQ    #16-1,D3     CONT. LOOP := 16 (-1 PER DBRA)
                           ;
0000400E: D080             LOOP      ADD.L    D0,D0        SHIFTA IL PRODOTTO A SIN. DI 1 BIT
00004010: D442                       ADD.W    D2,D2        SHIFTA MOLTIPLICAT. A SIN. DI 1 BIT
00004012: 6402                       BCC.S    STEP         SE MOLTIPLICATORE[15] ERA 1
                           ;
00004014: D081                       ADD.L    D1,D0        ...ALLORA SOMMA MOLTIPLICANDO
00004016: 51CB FFF6        STEP      DBRA     D3,LOOP      ...ALTRIMENTI CONTINUA
0000401A: 21C0 6004                  MOVE.L   D0,RESULT    SALVA RISULTATO
                           ;
0000401E: 4E75                       RTS

                                     END      PGM_8_3B
```

Diagramma di Flusso 8-3b

Differenza di velocità tra il programma 8-3a e il programma 8-3b

Questo programma esegue la stessa operazione di moltiplicazione a 16 bit del Programma 8-3a. Contando i cicli di clock delle due versioni si troverà che, a conferma di quanto detto in precedenza, la versione MULU richiede meno di 109 cicli, mentre la versione lunga (Programma 8-3b) richiede 58 cicli all'esterno del loop e 516 + 6n (n = numero di bit 1 del moltiplicatore) cicli all'interno.

8-4. Divisione binaria a 32 bit

Scopo: Dividere il numero a 32 bit privo di segno della variabile NUM1, alla locazione 6000, per il numero binario a 16 bit privo di segno della variabile NUM2, alla locazione 6004. Mettere il resto a 16 bit nella variabile REMAINDER, alla locazione 6006, e il quoziente a 16 bit nella variabile QUOTIENT, alla locazione 6008.

Problema Campione:

```
Input:   NUM1        - (6000) = 0074
                       (6002) = CBB1  dividendo a 32 bit
         NUM2        - (6004) = 0141  divisore a 16 bit

Output:  REMAINDER   - (6006) = 004C
         QUOTIENT    - (6008) = 5D25
                       o, in decimale, 7654321:321 = 23845
                       con il resto di 76
```

Programma 8-4

```
00006000:               DATA     EQU     $6000
00004000:               PROGRAM  EQU     $4000
                        ;
                                 ORG     DATA

00006000:               NUM1     DS.L    1            DIVIDENDO A 32 BIT
00006004:               NUM2     DS.W    1            DIVISORE A 16 BIT
00006006:               REMAIND  DS.W    1            RESTO A 16 BIT
00006008:               QUOTIENT DS.W    1            QUOZIENTE A 32 BIT
                        ;
                                 ORG     PROGRAM

00004000: 2038 6000     PGM_8_4  MOVE.L  NUM1,D0      DIVIDENDO A 32 BIT
00004004: 80F8 6004              DIVU    NUM2,D0      DIVISIONE SENZA SEGNO - NUM1/NUM2
00004008: 21C0 6006              MOVE.L  D0,REMAIND   SALVA RISULTATO- RESTO E QUOZIENTE
                        ;
0000400C: 4E75                   RTS

                                 END     PGM_8_4
```

Le istruzioni di divisione

L'MC68000 dispone di due istruzioni (DIVU e DIVS), che eseguono una divisione utilizzando un dividendo binario a 32 bit ed un divisore binario a 16 bit e dando origine ad un quoziente a 16 bit e ad un resto, anch'esso a 16 bit. L'istruzione DIVU è destinata all'aritmetica priva di segno, mentre l'istruzione DIVS viene impiegata con i numeri provvisti di segno. In quest'ultimo caso, la divisione dà un resto che ha lo stesso segno del dividendo; il segno del quoziente è

positivo se entrambi gli operandi hanno lo stesso segno e negativo se hanno segni diversi. Entrambe le istruzioni mettono il resto nei 16 bit più significativi del registro dati destinazione, mentre il quoziente viene posto nei 16 meno significativi.

Condizioni di errore nelle divisioni

Quando si eseguono l'una o l'altra delle due istruzioni di divisione si possono verificare due condizioni particolari. Innanzitutto, se il divisore è uguale a zero il processore genera una Trap da divisione per zero. (Per una descrizione delle Trap e del loro meccanismo, vi rimandiamo al Capitolo 15). In secondo luogo, il processore può rilevare una condizione di overflow e, di conseguenza, verrà posto a 1 il bit di Overflow (V) del registro di stato, mentre gli operandi non vengono modificati.

PROBLEMI

8-1. Sottrazione binaria in precisione multipla

Scopo: Sottrarre un numero multi-word da un altro. La lughezza di entrambi i numeri, espressa in word, è data dalla variabile LENGTH, alla locazione 6000. I numeri si trovano (i bit più significativi per primi) nelle variabili NUM1 e NUM2, alle locazioni 6002 e 6102, rispettivamente. Sottrarre il numero che si trova in NUM2 da quello in NUM1. Salvare il risultato in NUM1.

Problema Campione:

```
Input:    LENGTH    - (6000) = 0003
          NUM1      - (6002) = 2F5B
                      (6004) = 47C3
                      (6006) = 306C
          NUM2      - (6102) = 14DF
                      (6104) = 85B8
                      (6106) = 03BC

Output:   NUM1      - (6002) = 1A7B
                      (6004) = C20B
                      (6006) = 2CB0

Cioè:      2F5B47C3306C
          -14DF85B803BC
          -------------
           1A7BC20B2CB0
```

8-2. Sottrazione decimale

Scopo: Sottrarre un numero decimale (BCD) multi-byte da un altro. La lunghezza in byte di entrambi i numeri è nella variabile da un byte LENGTH, alla locazione 6000. I numeri (le cifre più significative per prime) si trovano nelle variabili NUM1 e NUM2, alle locazioni 6001 e 6101, rispettivamente. Sottrarre il numero contenuto in NUM2 da quello in NUM1. Salvare la differenza in NUM1.

Problema Campione:

Input:	LENGTH	– (6000)	= 04
	NUM1	– (6001)	= 36
		(6002)	= 70
		(6003)	= 19
		(6004)	= 85
	NUM2	– (6101)	= 12
		(6102)	= 66
		(6103)	= 34
		(6104)	= 59
Output:	NUM1	– (6001)	= 24
		(6002)	= 03
		(6003)	= 85
		(6004)	= 26

Cioè:

```
 36701985
-12663459
---------
 24038526
```

8-3. Moltiplicazione di 32 bit per 32 bit

Scopo: Moltilpicare il valore a 32 bit della variabile NUM1, che inizia nella locazione di memoria 6000 (ordine alto), per il valore a 32 bit della variabile NUM2, alla locazione 6004. Eseguire la moltiplicazione due volte: prima usare l'istruzione MULU e mettere il risultato nella variabile PROD1, che inizia alla locazione 6008; quindi, usare il metodo "somma e shift" , illustrato con il Programma 8-3b e mettere il risultato nella variabile a 64 bit PROD2, che inizia alla locazione 6010.

Problema Campione:

```
Input:   NUM1    - (6000)  = 0024
                   (6002)  = 68AC
         NUM2    - (6004)  = 0328
                   (6008)  = 1088

Output:  PROD1   - (6008)  = 0000
                   (600A)  = 72EC
                   (600C)  = BBC2
                   (600E)  = 5B60
         PROD2   - (6010)  = 0000
                   (6012)  = 72EC
                   (6014)  = B8C2
                   (6016)  = 5B60
```

BIBLIOGRAFIA

Altri metodi per la realizzazione di moltiplicazioni, divisioni ed altre funzioni aritmetiche sono trattati in:

Ali, Z. "Know the LSI Hardware Tradeoffs of Digital Signal Processors", *Electronic Design,* June 21, 1979, pp. 66-71.

Geist, D.J. "MOS Processor Picks up Speed with Bipolar Multipliers", *Electronics,* July 7, 1977, pp. 113-15.

Kolodzinski, A. and D. Wainland. "Multiplying with a Microcomputer," *Electronic Design,* January 18, 1978, pp. 78-83.

Mor, S. "An 8 x 8 Multiplier and 8-Bit Microprocessor Perform 16 x 16-Bit Multiplication", *EDN,* November 5, 1979, pp. 147-52.

Tao, T.F. et al. "Applications of Microprocessors in Control Problems," Proceedings of the 1977 Joint Automatic Control Conference, San Francisco, Ca., June 22-24, 1977.

Waser, S. "State-of-the Art in High-Speed Arithmetic Integrated Circuits," *Computer Design, July 1978, pp. 67-75.*

Waser, S. and A. Peterson. "Medium-Speed Multipliers Trim Cost, Shrink Bandwidth in Speech Transmission," Electronic Design, February 1, 1979, pp. 58-65.

Weissberger, A.J. and T. Toal "Tough Mathematical Tasks are Child's Play for Number Cruncher," *Electronics,* February 17, 1977, pp. 102-07.

CAPITOLO 9

TABELLE E LISTE

Situazioni in cui si impiegano tabelle o liste

Le tabelle e le liste sono due delle strutture di dati fondamentali usate da tutti gli elaboratori. Abbiamo già visto l'impiego di tabelle nelle conversioni di codice e nelle operazioni aritmetiche, ma le possiamo utilizzare anche per identificare e rispondere a dei comandi o a delle istruzioni, per accedere a determinati file o record, per definire il significato di tasti e interruttori oppure per scegliere fra routine differenti. Le liste sono, di solito, meno strutturate delle tabelle ed elencano particolari funzioni che il processore deve eseguire, eventuali messaggi o dati da utilizzare oppure una serie di condizioni variabili da tenere sotto controllo. Le tabelle rappresentano un ottimo metodo da adottare quando dobbiamo scegliere fra varie opzioni possibili o risolvere dei problemi, senza ricorrere a calcoli complessi o a delle funzioni logiche. Il compito del programmatore consiste unicamente nell'organizzare la tabella in modo tale da facilitare l'accesso ai vari elementi che la compongono. Le liste consentono l'esecuzione di varie funzioni successive, la preparazione di una serie di risultati e la realizzazione di insiemi di dati correlati fra loro (o data base). Gli esempi mostrano in che modo aggiungere o togliere elementi da una lista.

ESEMPI DI PROGRAMMAZIONE

9-1. Aggiunta di un elemento ad una lista

Scopo: Aggiungere il contenuto della variabile word ITEM, posta alla locazione di memoria 6000, ad una lista, a meno che non sia già presente. La lista è formata da elementi della lunghezza di una word ed il suo indirizzo iniziale si trova nella variabile long word LIST, alla locazione di memoria 6002. La prima word della lista ne contiene la lunghezza, espressa sempre in word.

Problemi Campione:

a. Input: ITEM – (6000) = 16B2
LIST – (6002) = 00005000 Ind. della lista
(5000) = 0004
(5002) = 5376
(5004) = 7618
(5006) = 138A
(5008) = 21DC

Output: (5000) = 0005 Lungh. della lista
.
.
.
(500A) = 16B2

b. Input: ITEM – (6000) = 16B2
LIST – (6002) = 0005000
(5000) = 0003
(5002) = 5376
(5004) = 16B2
(5006) = 7431

Output: Nessun cambiamento, in quanto l'elemento si trova già nella lista alla locazione 5004.

Diagramma di Flusso 9-1a

Progamma 9-1a:

```
00006000:                      DATA     EQU     $6000
00004000:                      PROGRAM  EQU     $4000
                               ;
00006000:                      ITEM     EQU     $6000         VALORE DA RICERCARE
00006002:                      LIST     EQU     $6002         PUNTATORE INIZIO LISTA
                               ;
                                        ORG     PROGRAM

00004000: 3038 6000            PGM_9_1A MOVE.W  ITEM,D0       PRENDI VALORE DA RICERCARE
00004004: 2078 6002                     MOVEA.L LIST,A0       A0 - PUNTATORE LISTA
00004008: 2248                          MOVEA.L A0,A1         SALVA PUNTATORE IN CONT. LISTA
0000400A: 3218                          MOVE.W  (A0)+,D1      D1.W - NUM. DI ELEMENTI IN LISTA
0000400C: 5341                          SUBQ.W  #1,D1         AGGIUSTA PER DBEQ
                               ;
0000400E: B058                 LOOP     CMP.W   (A0)+,D0      CONTROLLA SE ELEM. SEG. UGUALE
00004010: 57C9 FFFC                     DBEQ    D1,LOOP       CONTINUA FINCHE' NON TROVI ELEM.
                               ;                              UGUALE O FINE LISTA
00004014: 6704                          BEQ.S   DONE          SE UGUALE ALLORA VAI A DONE
                               ;
00004016: 3080                          MOVE.W  D0,(A0)       ...ALTRIMENTI AGGIUNGI IL
                               ;                              NUOVO ELEMENTO ALLA LISTA
00004018: 5251                          ADDQ.W  #1,(A1)       INCREMENTA CONT. LISTA
                               ;
0000401A: 4E75                 DONE     RTS

                                        END     PGM_9_1A
```

Spiegazioni del programma 9-1a

In questo programma ci serviamo dell'indirizzamento con autoincremento per accedere indirettamente alla lista attraverso il registro A0. Quando mettiamo la lunghezza della lista nel registro D1 viene incrementato automaticamente anche il puntatore in A0, per cui, quando ha inizio il LOOP, esso indica il primo elemento della lista. Quando usciamo dal ciclo, perchè i vari confronti hanno dato esito negativo, il puntatore è già stato incrementato e punta alla locazione successiva all'ultimo elemento della lista; perciò, non sarà necessario aggiustarlo per aggiungere un nuovo elemento alla fine di essa. È opportuno un confronto con il Programma 5-4b, per avere un quadro ancora più chiaro delle situazioni in cui è necessario o meno aggiornare i puntatori.

Chiaramente, il metodo che vi abbiamo appena mostrato si rivela scarsamente efficace nel caso di liste piuttosto lunghe. Possiamo migliorarlo limitando la ricerca ad una parte della lista o facendone l'ordinamento. Nel primo caso, ci serviremo del nuovo elemento per stabilire da quale punto iniziare la ricerca. È il metodo chiamato hashing , che ricorda molto da vicino quello seguito nella consultazione di un dizionario o di un elenco telefonico, utilizzando la prima lettera della parola cercata 1. Si potrebbero ordinare gli elementi della lista in base al loro valore numerico: la ricerca terminerebbe quando i valori elencati vanno oltre rispetto al nuovo ingresso (maggiori o minori, dipenderà dalla tenica di ordinamento adottata). Il nuovo elemento deve essere inserito in modo corretto, spostando verso il basso tutti gli altri elementi già presenti.

Ricerca con tecnica hashing

Il programma potrebbe essere ristrutturato in modo da utilizzare due tabelle, delle quali una sevirebbe per stabilire il punto di partenza all'interno dell'altra. Ad esempio, il punto di inizio della ricerca potrebbe essere stabilito in base alla cifra formata dai quattro bit più, o meno, significativi del nuovo elemento.

Limitazione del programma 9-1a e sua eliminazione

Il programma non funziona se la lunghezza della lista è zero. (Cosa accade?) Questo problema si può evitare effettuando un controllo iniziale della lunghezza. La procedura di inizializzazione e le altre modifiche necessarie sono indicate nel Programma 9-1b.

Programma 9-1b

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
00006000:                  ITEM      EQU      $6000        VALORE DA RICERCARE
00006002:                  LIST      EQU      $6002        PUNTATORE INIZIO LISTA
                           ;
                                     ORG      PROGRAM

00004000: 3038 6000        PGM_9_1B  MOVE.W   ITEM,D0      PRENDI VALORE DA RICERCARE
00004004: 2078 6002                  MOVEA.L  LIST,A0      A0 - PUNTATORE LISTA
00004008: 2248                       MOVEA.L  A0,A1        SALVA PUNTATORE IN CONTATORE LISTA
0000400A: 3218                       MOVE.W   (A0)+,D1     D1.W - NUM. DI ELEMENTI IN LISTA
0000400C: 670A                       BEQ.S    INSERT       SE LUNGH.=0 INSERISCI NUOVO ELEM.
                           ;
0000400E: 5341                       SUBQ.W   #1,D1        AGGIUSTA PER DBEQ
                           ;
00004010: B058             LOOP      CMP.W    (A0)+,D0     CONTROLLA SE ELEM.SEGUENTE UGUALE
00004012: 57C9 FFFC                  DBEQ     D1,LOOP      CONTINUA FINCHE' NON TROVI ELEM.
                           ;                               UGUALE O FINE LISTA
00004016: 6704                       BEQ.S    DONE         SE UGUALE ALLORA VAI A DONE
                           ;
00004018: 3080             INSERT    MOVE.W   D0,(A0)      ...ALTRIMENTI AGGIUNGI IL
                           ;                               NUOVO ELEMENTO ALLA LISTA
0000401A: 5251                       ADDQ.W   #1,(A1)      INCREMENTA CONT. LISTA
                           ;
0000401C: 4E75             DONE      RTS
                                     END      PGM_9_1B
```

Se la lunghezza della lista è zero significa che, al momento, non contiene nessun elemento; perciò, la variabile ITEM non può, comunque, essere presente e dovrà essere inserita in ogni caso (diventando il primo elemento).

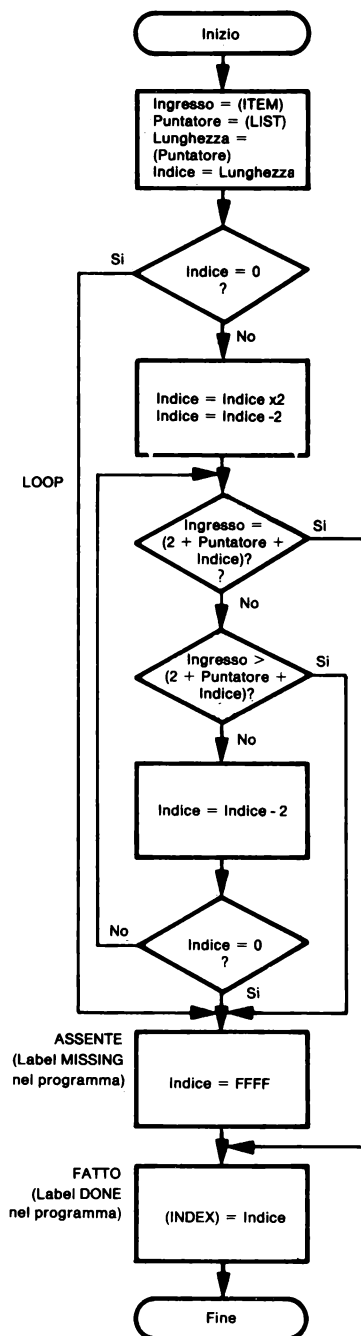
9-2. Controllo di una lista ordinata

Scopo: Controllare il contenuto della variabile a 16 bit ITEM, alla locazione di memoria 6000, e verificarne la presenza in una lista ordinata, formata da numeri binari a 16 bit privi di segno, in ordine crescente. L'indirizzo del primo elemento si trova nella variabile LIST, alla locazione 6004. Esso contiene la lunghezza della lista in word. Se il contenuto di ITEM si trova già nella lista, mettere l'indice relativo alla sua posizione nella variabile INDEX a 6002; altrimenti, assegnare ad INDEX il valore $FFFF_{16}$.

Problemi Campione:

```
a. Input:   ITEM    - (6000) = 5376
            LIST    - (6004) = 00005000 Lungh. della lista
                      (5000) = 0004
                      (5002) = 138A
                      (5004) = 21DC
                      (5006) = 5376
                      (5008) = 8613

   Output:  INDEX   - (6002) = 0004  poichè l'elemento
                                     cercato si trova alla
                                     loc.5006 = (5002 + 0004)

b. Input:   ITEM    - (6000) = 46B2
            LIST    - (6004) = 0005000
                      (5000) = 0002
                      (5002) = 138A
                      (5004) = 71DC
```

Output: INDEX – (6002) = FFFF poichè l'elemento cercato non compare nella lista.

Diagramma di Flusso 9-2a

Programma 9-2a:

```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
00006000:                ITEM     EQU      $6000
00006002:                INDEX    EQU      $6002
00006004:                LIST     EQU      $6004
                         ;
                                  ORG      PROGRAM

00004000: 3038 6000      PGM_9_2A MOVE.W   ITEM,D0          PRENDI VALORE DA RICERCARE
00004004: 2078 6004               MOVEA.L  LIST,A0          PRENDI IND. INIZIALE DELLA LISTA
00004008: 7200                    MOVEQ    #0,D1            AZZERA CONT. ELEMENTI
0000400A: 3210                    MOVE.W   (A0),D1          PRENDI CONT. ELEMENTI
0000400C: 6710                    BEQ.S    MISSING          SE LUNGH.=0, NON E' IN LISTA
                         ;
0000400E: D241                    ADD.W    D1,D1            OGNI ELEM. CONSISTE DI DUE BYTE
00004010: 5541                    SUBQ.W   #2,D1            VAR. INDICE = 0 - (LUNGH.*2-2)!
                         ;
00004012: B070 1002      LOOP     CMP.W    2(A0,D1.W),D0    RICERCA DALLA FINE ALL'INIZIO LISTA
00004016: 6708                    BEQ.S    DONE             VAL. IN LISTA, D1 CONTIENE INDICE
00004018: 6204                    BHI.S    MISSING          ELEM. PIU' PICCOLO,VAL.NON IN LISTA
0000401A: 5541                    SUBQ.W   #2,D1            INDICE PER SUCC. ELEM. PIU' PICCOLO
0000401C: 64F4                    BCC      LOOP             INDICE >= 0 - CONTINUA
                         ;
0000401E: 72FF           MISSING  MOVEQ    #$FF,D1          "NON TROVATO"-INDICE
                         ;
00004020: 31C1 6002      DONE     MOVE.W   D1,INDEX         SALVA INDICE
                         ;
00004024: 4E75                    RTS

                                  END      PGM_9_2A
```

La fase di ricerca si avvale, in questo caso, del fatto che i vari elementi sono disposti secondo un certo tipo di ordinamento. Iniziamo dall'ultimo elemento della lista, che è anche il più grande. Non appena troviamo un elemento minore rispetto al nuovo ingresso, la ricerca ha termine, dal momento che gli altri saranno, senz'altro, ancora più piccoli. Provate con un esempio, per convincervi della validità di questa procedura.

Come per il programma precedente, la scelta di un buon punto di partenza velocizzerà la ricerca. Un metodo valido potrebbe essere quello di iniziare dall'elemento centrale della lista, stabilire in quale delle due metà si trova il nuovo ingresso, quindi dividere quella metà in altre due e così via. È quella che viene chiamata ricerca binaria: infatti, ogni volta, dividiamo in due metà la parte rimanente della lista.[2 3]

Ricerca binaria

Spiegazione del programma 9-2a

Il Programma 9-2a funziona anche se la lunghezza della lista è zero, in quanto viene effettuato un controllo al momento di formare l'indice. Quello adottato con l'istruzione CMP.W è un tipico esempio di indirizzamento indicizzato con spostamento. Il registro indirizzi A0 punta alla "base" di una struttura dati, che , in questo caso, è una lista ordinata con il primo elemento che ne indica la lunghezza. Lo spostamento serve ad indirizzare un sottoinsieme, cioè il primo numero della lista. Il registro D1 è utilizzato come registro indice per accedere, dinamicamente, ai valori in essa contenuti. Questo metodo è illustrato nello schema seguente:

2(A0,D1)
disp(An,Dn)
Byte più significativo della lunghezza
Byte meno significativo della lunghezza
Primo elemento nella lista
Secondo elemento nella lista
Terzo elemento nella lista

Non dimenticate che lo spostamento viene interpretato come un numero in complemento a due, per cui è possibile avere anche dei valori negativi. Dal momento, che si tratta di un numero ad 8 bit provvisto di segno, sono possibili spostamenti in un intervallo compreso fra -128 e +127 byte.

L'indirizzo effettivo viene calcolato sommando lo spostamento (dopo averne esteso il segno) ai *32 bit* del registro indirizzi e del registro indice, il cui contenuto è considerato come un numero provvisto di segno. Se per il registro indice definiamo una grandezza pari ad una word, come abbiamo fatto con D1, il relativo valore viene trasformato in uno a 32 bit mediante l'estensione del segno, prima di calcolare l'indirizzo effettivo.

Dal momento che il registro indice può contenere un numero negativo, l'indirizzo effettivo finale può trovarsi prima o dopo l'indirizzo di base presente nel registro indirizzi.

In questo programma abbiamo confrontato dei valori privi di segno (BHI). Nel primo dei problemi campione, un confronto effettuato mediante BGT non funzionerebbe in modo corretto, poichè l'ultimo elemento della lista, 8613, ha il bit di segno a uno. Confronti fra valori privi di segno sono particolarmente utili quando si ha a che fare con degli indirizzi, che sono sempre senza segno.

Le due istruzioni di diramazione (BEQ.S e BHI.S), presenti in questo programma, possono essere sostituite da un' istruzione unica, che renderà più veloce l'esecuzione del ciclo. Lo vediamo nel Programma 9-2b:

Programma 9-2b

```
00006000:                DATA     EQU      $6000
00004000:                PROGRAM  EQU      $4000
                         ;
00006000:                ITEM     EQU      $6000
00006002:                INDEX    EQU      $6002
00006004:                LIST     EQU      $6004
                         ;
                                  ORG      PROGRAM

00004000: 3038 6000      PGM_9_2B MOVE.W   ITEM,D0             PRENDI VALORE DA RICERCARE
00004004: 2078 6004               MOVEA.L  LIST,A0             PRENDI IND. INIZIALE DELLA LISTA
00004008: 7200                    MOVEQ    #0,D1               AZZERA CONT. ELEMENTI
0000400A: 3210                    MOVE.W   (A0),D1             PRENDI CONT. ELEMENTI
0000400C: 6710                    BEQ.S    MISSING             SE LUNGH.=0, NON E' IN LISTA
                         ;
0000400E: D241                    ADD.W    D1,D1               OGNI ELEM. CONSISTE DI DUE BYTE
00004010: 5541                    SUBQ.W   #2,D1               VAR. INDICE = 0 - (LUNGH.*2-2)!
                         ;
00004012: B070 1002      LOOP     CMP.W    2(A0,D1.W),D0       RICERCA DALLA FINE ALL'INIZIO LISTA
00004016: 6404                    BCC.S    LPEXIT              FATTO SE TROVATO O VAL.>ELEM.LISTA
00004018: 5541                    SUBQ.W   #2,D1               INDICE PER SUCC. ELEM. PIU' PICCOLO
0000401A: 64F6                    BCC      LOOP                INDICE >= 0 - CONTINUA
                         ;
0000401C: 6702           LPEXIT   BEQ.S    DONE                VAL. IN LISTA, D1 CONTIENE INDICE
                         ;
0000401E: 72FF           MISSING  MOVEQ    #$FF,D1             "NON TROVATO"-INDICE
                         ;
00004020: 31C1 6002      DONE     MOVE.W   D1,INDEX            SALVA INDICE
                         ;
00004024: 4E75                    RTS

                                  END      PGM_9_2B
```

In questo caso, la prima istruzione di diramazione, all'interno del loop, trasferisce il controllo a LPEXIT, se il nuovo ingresso è uguale o maggiore rispetto all'elemento della lista con cui viene confrontato. Esiste la possibilità che si verifichi una condizione pericolosa in un caso come questo. Osservate l'istruzione BEQ a LPEXIT. Il programma può arrivare a questa istruzione in due modi diversi:

1. l'istruzione BCC.S LPEXIT, all'interno del ciclo provoca una diramazione a LPEXIT. In questo caso, i flag di stato rispecchiano il risultato dell'istruzione CMP.W all'interno del ciclo.
2. se tutti gli elementi della lista sono stati controllati senza trovarne nessuno uguale al nuovo ingresso, allora il loop termina e viene eseguita l'istruzione immediatamente successiva a BCC LOOP. In questo caso, i flag di stato sono modificati sulla base del risultato dell'istruzione SUBQ all'interno del ciclo

Quindi, l'istruzione BEQ a LPEXIT controlla i flag di stato che sono modificati da una delle due istruzioni precedenti. Bisogna accertarsi che non ci siano condizioni di conflittualità che diano luogo a risultati inattesi o ad errori non facilmente individuabili. Il modo più sicuro per evitare errori è di costruire una tabella in modo da verificare cosa accade in tutte le situazioni possibili. Nel caso del Programma 9-2b potremmo usare una tabella come questa:

Spiegazione del programma 9-2b

		N	Z	V	C	
Dopo CMP.W	item < (list)	?	0	?	1	Questo dovrebbe causare l'uscita dal loop. Usa BCC per uscire
	item = (list)	0	1	0	0	
	item > (list)	?	0	?	0	
Dopo SUBQ	D1 $\geq$ 0	?	?	?	0	Questo dovrebbe far terminare il loop. Usa BCC per continuare il loop
	D1 = −2	1	0	0	1	

Come si può rilevare da questa tabella, il flag Z sarà sempre 0 alla fine del ciclo. Perciò, quando viene eseguita l'istruzione BEQ a LPEXIT, in seguito all'istruzione BCC LOOP, non si verifica la diramazione a DONE.

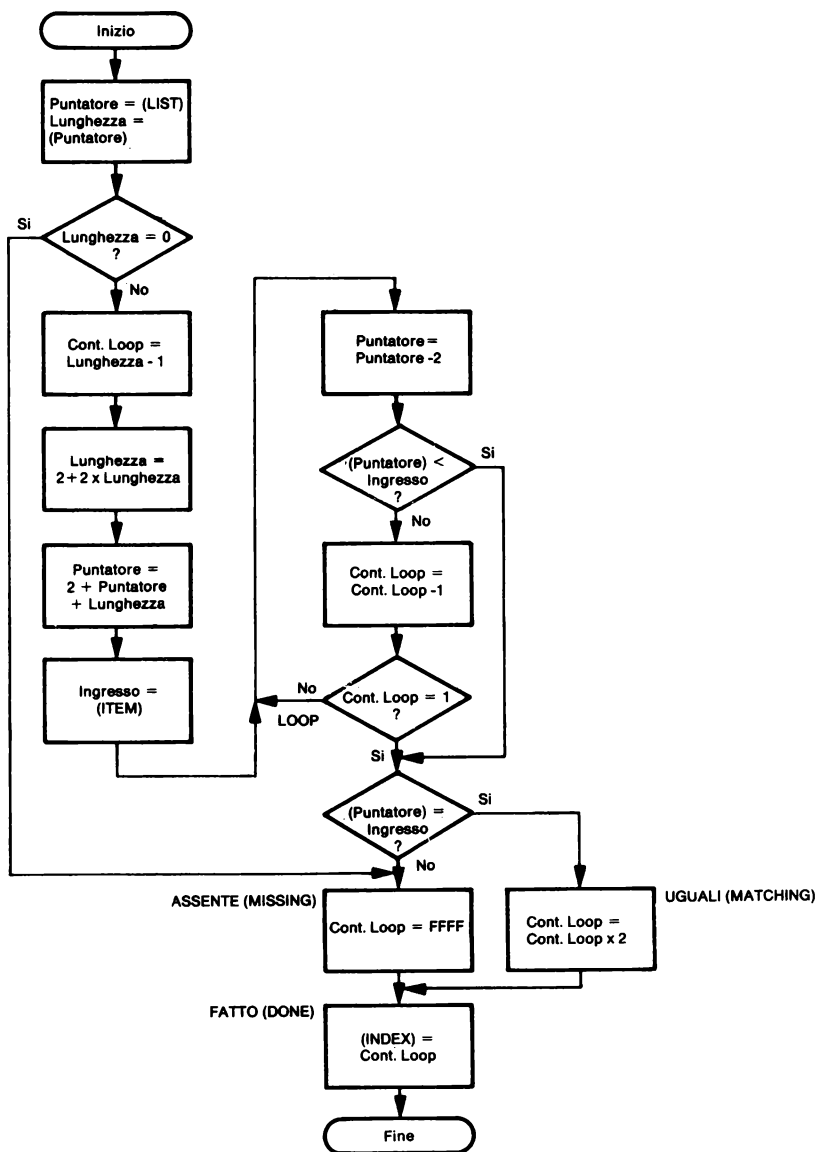
Come velocizzare il programma 9-2b

Possiamo rendere leggermente più veloce questo programma, utilizzando, all'interno del loop, l'istruzione CMP con predecremento e l'istruzione DBcc. Le modifiche necessarie sono indicate nel Programma 9-2c.

Programma 9-2c

```
00006000:                DATA      EQU      $6000
00004000:                PROGRAM   EQU      $4000
                         ;
00006000:                ITEM      EQU      $6000
00006002:                INDEX     EQU      $6002
00006004:                LIST      EQU      $6004
                         ;
                                   ORG      PROGRAM

00004000: 2078 6004      PGM_9_2C  MOVEA.L  LIST,A0         PRENDI IND. INIZIALE DELLA LISTA
00004004: 3210                     MOVE.W   (A0),D1         PRENDI CONT. ELEMENTI
00004006: 6718                     BEQ.S    MISSING         SE LUNGH.=0, NON E' IN LISTA
                         ;
00004008: 5341                     SUBQ.W   #1,D1           CORREZ. PER DBCC E VAR. INDICE
0000400A: 3401                     MOVE.W   D1,D2           D2 E' IL CONTATORE DEL LOOP
                         ;
0000400C: D241                     ADD.W    D1,D1           OGNI ELEM. CONSISTE DI DUE BYTE
0000400E: 5441                     ADDQ.W   #2,D1           CORREZ. PER IL PREDECR. DEL CICLO
00004010: 41F0 1002                LEA      2(A0,D1.W),A0   PUNTATORE OLTRE FINE LISTA
                         ;
00004014: 3038 6000                MOVE.W   ITEM,D0         PRENDI VALORE DA RICERCARE
                         ;
00004018: B060           LOOP      CMP.W    -(A0),D0        INIZIA DALLA FINE DELLA LISTA
0000401A: 54CA FFFC                DBCC     D2,LOOP         CONTROLLA SE ELEM.>VAL. E SE
                         ;                                  SONO RIMASTI ALTRI ELEMENTI
0000401E: 6704                     BEQ.S    MATCHING        VALORE IN LISTA, D2 HA L'INDICE
                         ;
```

```
00004020: 74FF              MISSING  MOVEQ   #$FF,D2        "NON TROVATO" - INDICE
00004022: 6002                       BRA.S   DONE
                            ;
00004024: D442              MATCHING ADD.W   D2,D2          AGGIUSTA INDICE PER UNA WORD
00004026: 31C2 6002         DONE     MOVE.W  D2,INDEX       SALVALO
                            ;
0000402A: 4E75                       RTS

                                     END     PGM_9_2C
```

Diagramma di Flusso 9-2c

Spiegazione del programma 9-2c

Oltre a cambiare il ciclo, abbiamo apportato alcune piccole modifiche alla fase di inizializzazione. Prima di tutto, l'oggetto della ricerca (il nuovo ingresso) non viene prelevato finchè non abbiamo controllato se la lunghezza della lista è zero. È inutile disporre del nuovo elemento, se poi la ricerca non ha luogo.

L'istruzione LEA serve ad ottenere l'indirizzo del primo elemento della struttura dati, analogamente a quanto acccade nei Programmi 9-2a e 9-2b, ma, in questo caso, l'indirizzo di partenza è costruito prima dell'inizio del ciclo.

Osservate, anche, come il Programma 9-2c eviti il problema del flag di stato, cui abbiamo accennato a proposito del Programma 9-2b. Dato che l'istruzione DBcc non influisce sui codici di condizione, al termine del loop essi hanno ancora i valori determinati dall'istruzione CMP.W, che siamo liberi di controllare a nostro piacimento.

Differenza di velocità tra i programmi 9-2c e 9-2a

Confrontando i cicli di clock necessari all'esecuzione del loop del Programma 9-2c, vi renderete conto che è due volte più veloce di quello del Programma 9-2a. Qualora esista la possibilità che un loop sia ripetuto molte volte, vale la pena cercare di ridurne la durata.

Il tempo medio di esecuzione di questa tecnica di ricerca semplice, indipendentemente da quale dei tre programmi utilizziate, aumenta in modo lineare con la lunghezza della lista, mentre il tempo medio di una ricerca binaria aumenta in modo logaritmico. Ad esempio, se la lunghezza della lista viene raddoppiata, la prima tecnica richiede un tempo doppio, mentre il metodo della ricerca binaria necessita di una sola iterazione in più.

9-3. Rimuovere un elemento da una coda

Scopo: La variabile QUEUE, alla locazione di memoria 6000, contiene l'indirizzo iniziale di una coda (queue). Salvare l'indirizzo del primo elemento (inizio) della coda nella variabile POINTER, alla locazione 6002. Modificare la coda, rimuovendo il primo elemento. Ogni elemento ha la grandezza di una word e contiene l'indirizzo dell'elemento successivo; l'ultimo contiene uno zero per indicare la fine della coda.

Le strutture a coda

Le code contengono, in genere, dei dati nello stesso ordine in cui dovranno essere utilizzati oppure delle funzioni nell'ordine in cui saranno eseguite. La coda è una struttura di tipo FIFO (first-in first-out): il primo dato che entra è anche il primo ad uscire, in quanto i vari elementi vengono rimossi nello stesso ordine in cui sono stati introdotti. I sistemi operativi si servono delle code per inserirvi una serie di funzioni, che devono essere eseguite in sequenza. I driver di I/O trasferiscono da e verso delle code quei dati che devono essere trasmessi o manipolati in un ordine ben preciso. Una struttura a coda è impiegata anche nel caso dei buffer, in modo da poter

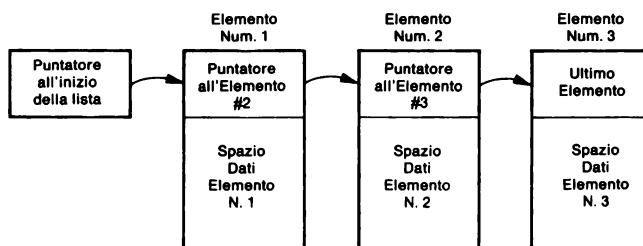
localizzare più facilmente il successivo buffer disponibile. Le code servono, inoltre, per collegare richieste di memoria, di temporizzazione o di I/O, avendo sempre la certezza che, in questo modo, saranno soddisfatte nell'ordine voluto.

Nelle applicazioni reali, ogni elemento di una coda contiene, di solito, una notevole quantità di informazioni e/o di spazio per la memorizzazione di dati, oltre all'indirizzo che indica l'inizio dell'elemento successivo.

Liste Collegate (Linked List)

Esempio di implementazione di una coda

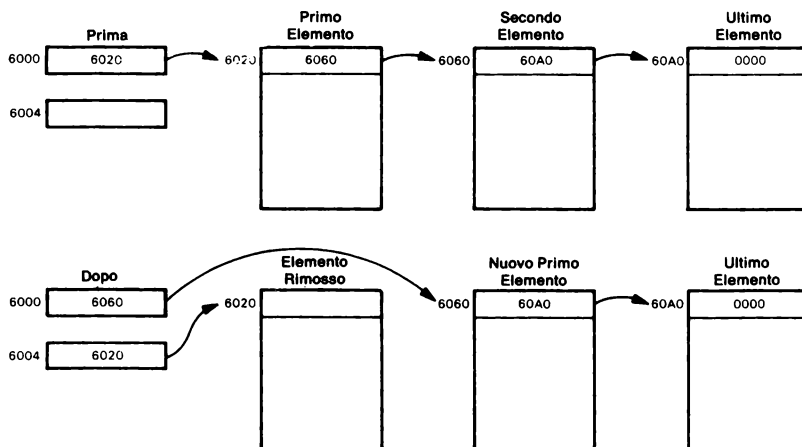
Un modo per realizzare una coda è quello di fare uso di una lista collegata. Notate che esiste una differenza fra le strutture dati e la loro realizzazione. Ad esempio, una coda è una struttura dati per realizzare la quale esistono molti modi diversi. La funzione fondamentale (first-in, first-out) resta, comunque, sempre la stessa, indipendentemente dalle modalità di realizzazione.

Il principio fondamentale di una lista collegata è il fatto che ciascun elemento deve contenere l'indirizzo di quello successivo, oltre ad altri eventuali dati propri di quel particolare elemento. Possiamo rappresentarla nel modo seguente:

Principale vantaggio delle liste collegate

Il vantaggio di questa tecnica è che gli elementi della lista non devono essere memorizzati sequenzialmente, dal momento che ciascun elemento contiene l'indirizzo che indica la posizione di quello successivo. Per cambiare l'ordine di due elementi tutto quello che dobbiamo fare è cambiare i puntatori, senza spostare i dati. Così per rimuovere il primo elemento di una coda, basta spostare una coppia di puntatori ed il gioco è fatto; non dobbiamo spostare un solo dato, ma soltanto un paio di indirizzi. Le liste collegate richiedono più memoria rispetto a quelle sequenziali, ma è molto più semplice aggiungere, togliere o inserire degli elementi.

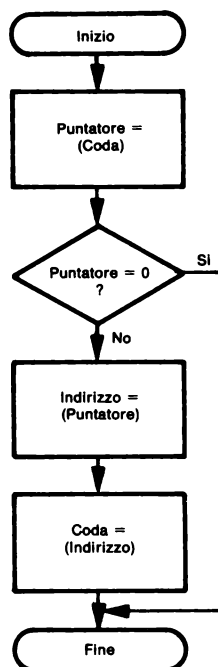
Problemi Campione:

```
a. Input:   QUEUE    - (6000) = 00006020 Ind. del primo ele-
                                         mento della coda
                     - (6020) = 00006060 Primo elemento
                       (6060) = 000060A0
                       (60A0) = 00000000 Ultimo elemento
```

Output: QUEUE – (6000) = 00006060 Indirizzo del nuovo primo elemento
POINTER – (6004) = 00006020 Ind. dell'elemento rimosso dalla coda

b. Input: QUEUE – (6000) = 00000000 Coda vuota

Output: QUEUE – (6000) = 0000

POINTER – (6004) = 0000 Nessun elemento disponibile

Diagramma di Flusso 9-3

Programma 9-3

```
00006000:                  DATA     EQU      $6000
00004000:                  PROGRAM  EQU      $4000
                           ;
00006000:                  QUEUE    EQU      $6000            INDIRIZZO INIZIO CODA
00006004:                  POINTER  EQU      $6004            IND. PRECEDENTE INIZIO CODA
                           ;
                                    ORG      PROGRAM

00004000: 21F8 6000
00004004: 6004             PGM_9_3  MOVE.L   QUEUE,POINTER    SALVA PRECEDENTE INIZIO CODA
00004006: 6708                      BEQ.S    DONE             SE CODA VUOTA VAI A DONE
                           ;
00004008: 2078 6004                 MOVE.L   POINTER,A0       ..ALTRIMENTI RIMUOVI PRIMO ELEM.
0000400C: 21D0 6000                 MOVE.L   (A0),QUEUE       E SOSTITUISCILO COL SECONDO
00004010: 4E75             DONE     RTS

                                    END      PGM_9_3
```

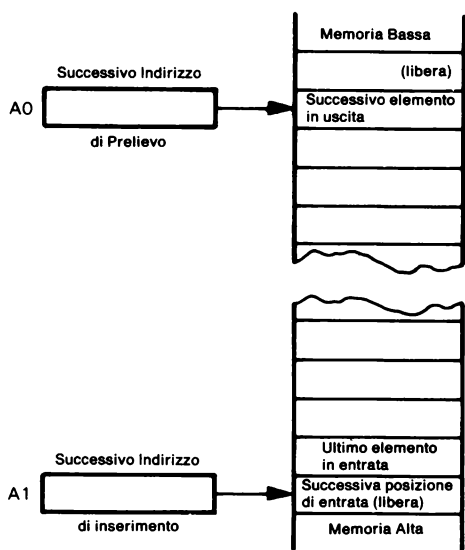
Liste Doppiamente Collegate

Alcune volte può essere necessario disporre di puntatori in entrambe le direzioni ed, in questo caso, ciascun elemento di una coda dovrà contenere gli indirizzi sia dell'elemento precedente che di quello successivo.[4,5] Queste liste doppiamente collegate consentono di rintracciare con facilità le varie fasi di una sequenza (ad es., ripetere la funzione precedente, se in quella attuale si è verificato un errore) o di accedere ai vari elementi da entrambe le estremità, indifferentemente (ad es., permettendo di rimuovere o cambiare gli ultimi due elementi senza dover esaminare l'intera coda). **Una struttura di questo tipo può, quindi, essere usata sia nel modo first-in, first-out che in quello last-in, first-out (l'ultimo ad entrare è il primo ad uscire), a seconda che i nuovi elementi siano aggiunti dall'estremità iniziale o da quella finale.**

Coda Vuota

Se nella coda non ci sono elementi, il programma azzera POINTER, alla locazione 6004. Un programma che preleva un elemento da una coda deve controllare questa locazione di memoria per vedere se la sua richiesta può essere soddisfatta (cioè, se in quella coda c'è veramente qualcosa). Potreste suggerire altri modi con cui indicare al programma richiedente se la coda è vuota o meno?

Implementazione alternativa di una coda

Un altro metodo per la realizzazione di una coda è una lista che occupa un certo numero di locazioni consecutive della memoria. L'architettura dell'MC68000 ben si adatta alla manipolazione di questo tipo di code, per la cui realizzazione possiamo utilizzare due qualsiasi dei registri indirizzi (A0 - A6) e l'indirizzamento con postincremento o predecremento, a seconda che la coda si accresca, rispettivamente, dalla parte bassa della memoria verso quella alta o viceversa. Lo schema seguente mostra una coda che si estende dal basso verso l'alto ed è realizzata mediante i registri indirizzi A0 e A1:

A0 contiene il puntatore al primo (o più vecchio) elemento della coda, mentre A1 punta alla locazione dove avverrà il successivo ingresso. Se si accede a questa coda servendoci dell'indirizzamento con postincremento, allora A0 conterrà sempre l'indirizzo del successivo elemento da prelevare, mentre A1 quello della locazione dove inserire il nuovo elemento.

Lo Stack

Un'altra struttura dati simile alla coda è lo stack, una lista di tipo LIFO (last-in, first-out: l'ultimo a entrare è il primo ad uscire). La maggior parte dei microprocessori dispone di speciali istruzioni di inserimento e rimozione, per manipolare gli stack; nell'MC68000 si possono usare le potenti istruzioni MOVE, unitamente all'indirizzamento con postincremento o predecremento.

La realizzazione di uno stack richiede un unico registro indirizzi e l'indirizzamento con postincremento o predecremento. Lo stesso processore, infatti, si serve del registro indirizzi A7 per gestire due stack speciali, quello di Sistema e quello Utente. Entrambi verranno descritti nel Capitolo 10.

L'Uso delle Strutture di Dati

I vari tipi di indirizzamento indicizzato e indiretto ci permettono di utilizzare le strutture dati in modo molto flessibile. Se, ad esempio, un

registro indirizzi contiene l'indirizzo iniziale di un blocco di informazioni, è possibile accedere ai singoli elementi mediante valori di spostamento costanti.

Quando possiamo utilizzare questo tipo di strutture? Ad esempio, quando vogliamo che un'apparecchiatura di controllo esegua una determinata serie di test. Tramite un pannello di controllo costruiamo una coda di blocchi di informazioni, uno per ogni test che l'operatore vorrà eseguire. Ogni blocco conterrà:

1. L'indirizzo iniziale del blocco successivo (oppure 0 se non c'è un blocco successivo)
2. L'indirizzo iniziale del programma che esegue il test.
3. L'indirizzo del dispositivo di input (ad es., una tastiera, un lettore di schede o una linea di comunicazione), dal quale verranno letti i dati nel corso del test.
4. L'indirizzo del dispositivo di output (ad es., una stampante, un terminale CRT o una linea di comunicazione), al quale verranno inviati i risultati del test.
5. Il numero di volte che il test deve essere ripetuto.
6. L'indirizzo iniziale dell'area di memoria destinata alla memorizzazione temporanea dei dati.
7. Un flag che indica se un test non riuscito debba o meno impedire l'esecuzione dei test successivi.

Naturalmente, un blocco può contenere anche altre informazioni qualora sia necessario specificare altre opzioni durante l'allestimento della procedura di controllo. Osservate come alcuni elementi di un blocco contengano dei dati, altri, invece, degli indirizzi, mentre altri ancora possano essere dei flag di 1 bit.

Precisiamo meglio cosa intendiamo in questo esempio quando parliamo di flessibilità. Ecco alcune delle procedure che utilizzando il metodo appena descritto sono disponibili per l'operatore finale:

1. Eseguire lo stesso test con differenti dispositivi di ingresso/uscita. Un'esecuzione di prova può utilizzare dati provenienti da una tastiera ed inviare i risultati ad un terminale video, mentre un'esecuzione nel corso di un normale ciclo produttivo si servirà di dati inviati tramite una linea telefonica e fornirà il risultato su una stampante.
2. Eseguire i vari test in un ordine qualsiasi, cambiandone soltanto la posizione all'interno della coda.
3. Mettere dei dati temporanei in una zona da dove possano essere facilmente visualizzati o richiamati da un programma di debugging.
4. Scegliere fra una serie di possibilità: interrompere i test, indicare eventuali cause d'errore oppure ripetere certe procedure. Anche in questo caso abbiamo la possibilità di usare durante le esecuzioni di prova ed il debugging del sistema opzioni diverse da quelle che verranno usate durante il normale ciclo produttivo.

5. Cancellare oppure inserire dei test semplicemente cambiando i puntatori che collegano un test a quello successivo. Un operatore può, in questo modo, correggere degli errori o apportare delle modifiche senza dover riscrivere l'intero elenco dei test.

Ad esempio, supponiamo che l'operatore inserisca la sequenza TEST 1, TEST 2, TEST 4 e TEST 5, tralasciando, per errore, TEST 3. I blocchi saranno collegati nel modo seguente:

Il blocco 1 (per il TEST 1) contiene l'indirizzo iniziale del blocco 2 (per il TEST 2).
Il blocco 2 (per il TEST 2) contiene l'indirizzo iniziale del blocco 3 (per il TEST 4).
Il blocco 3 (per il TEST 4) contiene l'indirizzo iniziale del blocco 4 (per il TEST 5).
Il blocco 4 (per il TEST 5) contiene un puntatore zero per indicare che si tratta dell'ultimo blocco.

L'inserimento del TEST 3 fra il TEST 2 e il TEST 4 richiede solamente le modifiche seguenti:

Il blocco 2 (per il TEST 2) deve contenere adesso l'indirizzo iniziale del blocco 5 (per Il TEST 3).
Il blocco 5 (per il TEST 3) deve contenere adesso l'indirizzo iniziale del blocco 3 (per Il TEST 4).

Non sono necessarie altre modifiche e nessun blocco deve essere spostato. Notate come sia molto più semplice inserire o cancellare facendo uso di liste collegate, anzichè di liste contenute in locazioni di memoria consecutive. Nel primo caso, infatti, non esiste il problema di spostare i vari elementi per creare o rimuovere degli spazi vuoti.

9-4. Ordinamento di numeri ad 8 bit

Scopo: Ordinare una lista di numeri binari ad 8 bit, privi di segno. L'indirizzo iniziale della lista è nella variabile LIST, alla locazione 6000. Il primo elemento indica il numero degli elementi restanti (cioè la lunghezza della lista escluso il primo elemento). In questo modo, la lista può contenere fino ad un massimo di 255 elementi (poiché 255_{10} è il massimo numero esprimibile con 8 bit).

Inserimento di un elemento in una lista collegata

Problema Campione:

Input:	LIST	– (6000) =	00005000	Ind. inizio lista
		(5000) =	06	Num. degli elementi
		(5001) =	2A	Primo elemento

		(5002) =	B5	
		(5003) =	60	
		(5004) =	3F	
		(5005) =	D1	
		(5006) =	19	
Output:	LIST	(6000) =	00005000	
		(5000) =	06	Num. degli elementi
		(5001) =	D1	Elemento più grande
		(5002) =	B5	
		(5003) =	60	
		(5004) =	3F	
		(5005) =	2A	
		(5006) =	19	Elemento più piccolo

Un semplice algoritmo di ordinamento

Una semplice tecnica di ordinamento funziona nel modo seguente:

Fase 1. Azzerare un flag chiamato EXCHANGE
Fase 2. Esaminare ciascuna coppia di numeri consecutivi. Se non sono ordinati, scambiarli fra loro e porre a 1 EXCHANGE.
Fase 3. Se EXCHANGE vale 1, dopo che tutta la lista è stata esaminata, ritornare alla fase 1.

EXCHANGE viene posto a uno se una qualunque coppia di numeri consecutivi non si trova nell'ordine desiderato; perciò, se EXCHANGE è zero, alla fine di un passaggio attraverso l'intera lista, significa che essa è completamente ordinata.

Algoritmo "bubble-sort"

Questo metodo di ordinamento viene chiamato "bubble sort" (ordinamento a bolle) ed è un algoritmo facilmente realizzabile, anche se piuttosto lento. Quando si tratta di ordinare delle liste lunghe e la velocità diventa importante vanno prese in considerazione tecniche diverse.[6-8]

In un caso piuttosto semplice questa tecnica agisce nel modo seguente. Supponiamo di dover ordinare in senso decrescente una lista contenente quattro elementi: 12, 03, 15, 08.

Prima Iterazione:

Fase 1. EXCHANGE = 0
Fase 2. La disposizione finale della serie è:
12
15
08
03
poichè la seconda coppia è scambiata (03, 15) e lo è anche la terza (03,08).
EXCHANGE = 1

Seconda Iterazione:

Fase 1. EXCHANGE = 0

Fase 2. La disposizione finale della serie è:

15
12
08
03

poichè la prima coppia (12, 15) è stata scambiata.
EXCHANGE = 1

Terza Iterazione:

Fase 1. EXCHANGE = 0

Fase 2. Gli elementi sono già in ordine, per cui non sono necessarie modifiche ed EXCHANGE rimane 0.

Questo metodo richiede sempre una iterazione in più per verificare che gli elementi siano nell'ordine voluto. Poichè l'ultima iterazione non effettua degli scambi, in realtà serve a poco. **Osservando gli esempi, vi renderete conto che molti dei confronti sono inutili e ripetitivi. Perciò questo metodo può essere notevolmente migliorato, soprattutto se il numero degli elementi è dell'ordine di migliaia o di milioni, come comunemente avviene per le grosse applicazioni. Nuove tecniche di ordinamento sono attualmente oggetto di ricerca[9].**

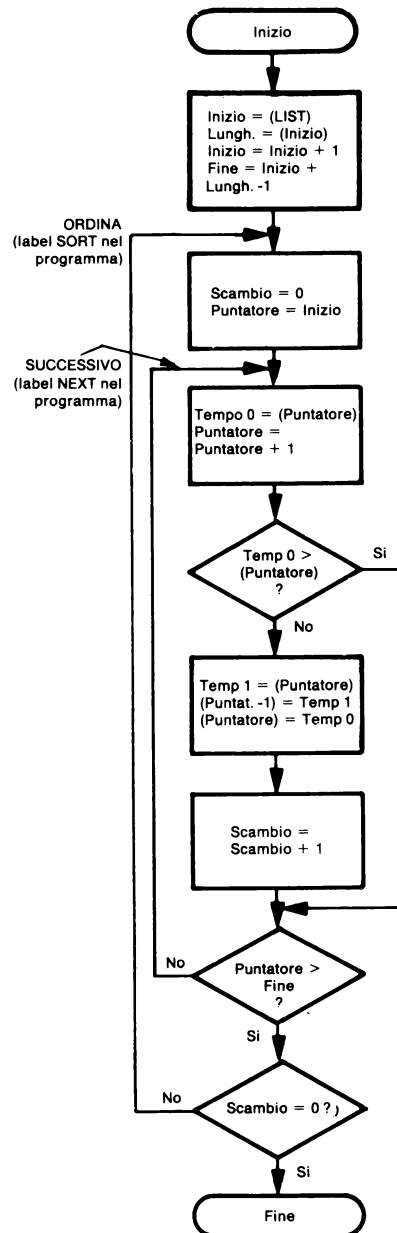
Listato Programma 9-4a

```
00006000:                    DATA      EQU      $6000
00004000:                    PROGRAM   EQU      $4000
                             ;
00006000:                    LIST      EQU      $6000             INDIRIZZO INIZIO LISTA
                             ;
                                       ORG      PROGRAM

00004000: 2078 6000          PGM_9_4A  MOVEA.L  LIST,A0           PUNTATORE INIZIO LISTA
00004004: 4240                         CLR.W    D0
00004006: 1018                         MOVE.B   (A0)+,D0          LUNGHEZZA DELLA LISTA
00004008: 43F0 00FF                    LEA      -1(A0,D0.W),A1    PUNT. ULTIMO ELEMENTO IN LISTA
                             ;
0000400C: 4241               SORT      CLR.W    D1                CONTATORE SCAMBI
0000400E: 2448                         MOVEA.L  A0,A2             PUNTATORE INIZIO LISTA
                             ;
00004010: 101A               NEXT      MOVE.B   (A2)+,D0          PRENDI ELEMENTO SUCCESSIVO
00004012: B012                         CMP.B    (A2),D0           CONFRONTALO CON ELEM. SEGUENTE
00004014: 640A                         BCC.S    NSWITCH           SE ELEM. PREC. >= VAI A NEXT
                             ;
00004016: 1212                         MOVE.B   (A2),D1           ..ALTRIMENTI SCAMBIA ELEMENTI
00004018: 1541 FFFF                    MOVE.B   D1,-1(A2)
0000401C: 1480                         MOVE.B   D0,(A2)
0000401E: 5241                         ADDQ.W   #1,D1             INCREMENTA CONTATORE SCAMBI
                             ;
00004020: B3CA               NSWITCH   CMPA.L   A2,A1             FINE DELLA LISTA
00004022: 62EC                         BHI      NEXT              SE NO, CONTROLLA ELEMENTO SUCC.
00004024: 4A41                         TST.W    D1                SI E' VERIFICATO UNO SCAMBIO?
00004026: 66E4                         BNE      SORT              SI', CONTINUA ORDINAMENTO
00004028: 4E75                         RTS

                                       END      PGM_9_4A
```

Spiegazione del programma 9-4a

Il programma deve diminuire di 1 il puntatore A1 perchè all'ultimo elemento, ovviamente, non ne seguono altri. L'ultimo confronto è tra il penultimo e l'ultimo elemento. Prima di iniziare ogni passaggio, non dobbiamo dimenticarci di inizializzare di nuovo il puntatore ed il flag di Exchange.

Gli esempi precedenti si servivano di contatori per controllare un loop mentre, in questo caso, confrontiamo degli indirizzi per cui non è necessario incrementare un contatore ad ogni passaggio. È interes-

Diagramma di Flusso 9-4

sante notare cosa accade se nella lista ci sono meno di due elementi. Il risultato non è corretto, sebbene la cosa non sia così tragica come sarebbe accaduto usando dei contatori. In realtà, è molto semplice prevenire una simile eventualità: basta inserire BRA.S NSWITCH prima dell'istruzione contrassegnata dalla label NEXT.

Due elementi uguali non devono essere scambiati; se lo fossero, lo scambio avverrebbe ad ogni passaggio e il programma non finirebbe mai.

Esistono molti modi per codificare questo programma di ordinamento a bolle mediante il set di istruzioni dell'MC68000. L'uso delle istruzioni di confronto da memoria a memoria riduce la lunghezza del programma ed ottimizza la fase di elaborazione del ciclo. Questa modifica, insieme ad altre, la trovate nel Programma 9-4b. Quali sono i vantaggi e gli svantaggi derivanti dall'uso di quelle istruzioni che agiscono su un singolo bit, per azzerare o mettere a 1 il flag indicante lo scambio? Cosa accade se non verifichiamo preventivamente l'eventuale assenza di elementi nella lista? Ricordate che DBRA controlla se il valore del contatore è uguale a -1.

Programma 9-4b

```
00006000:                     DATA      EQU      $6000
00004000:                     PROGRAM   EQU      $4000
                              ;
00006000:                     LIST      EQU      $6000         INDIRIZZO INIZIO LISTA
                              ;
                                        ORG      PROGRAM
00004000: 2078 6000           PGM_9_4B  MOVEA.L  LIST,A0       PUNTATORE LUNGHEZZA LISTA
00004004: 4280                          CLR.L    D0            AZZERA TUTTI I 32 BIT DI D0
00004006: 1018                          MOVE.B   (A0)+,D0      LUNGHEZZA DELLA LISTA
00004008: 6724                          BEQ.S    DONE          SE LUNGH. LISTA = 0 VAI A DONE
0000400A: 43E8 0001                     LEA      1(A0),A1      PUNT. AL SECONDO ELEMENTO
                              ;
0000400E: 0881 0000                     BCLR     #0,D1         FLAG DI SCAMBIO := 0
00004012: 5340                          SUBQ.W   #1,D0         AGGIUSTA CONT. PER L'ISTR. DBCC
00004014: 600E                          BRA.S    NSWITCH       CONTROLLA SE C'E' UN SOLO ELEM.
                              ;
00004016: B308                NEXT      CMPM.B   (A0)+,(A1)+   CONFORNTA ELEMENTI ADIACENTI
00004018: 630A                          BLS.S    NSWITCH       SE PRIMO <= SECONDO NON SCAMBIARLI
0000401A: 1420                          MOVE.B   -(A0),D2      SCAMBIA
0000401C: 10E1                          MOVE.B   -(A1),(A0)+   ...ELEMENTI
0000401E: 12C2                          MOVE.B   D2,(A1)+
00004020: 08C1 0000                     BSET     #0,D1         PONI A 1 FLAG DI SCAMBIO
                              ;
00004024: 51C8 FFF0           NSWITCH   DBRA     D0,NEXT       CONFRONTA TUTTI GLI ELEMENTI
00004028: 0801 0000                     BTST     #0,D1         FLAG DI SCAMBIO = 1?
0000402C: 66D2                          BNE      PGM_9_4B      SE SI', RIPETI IL CONTROLLO
                              ;
0000402E: 4E75                DONE      RTS
                                        END      PGM_9_4B
```

Sono stati scritti interi volumi sulle tecniche di ordinamento e di ricerca, perciò una loro trattazione va oltre gli scopi di questo libro. C'è, tuttavia, un aspetto che dobbiamo considerare. Alla fine di ogni passaggio, sappiamo che l'elemento più piccolo è in fondo alla lista; perciò, ogni volta, il numero delle coppie da confrontare diminuisce di uno. (Provate a rendervene conto osservando alcuni esempi. Riuscite a capire a cosa deve il suo nome questo metodo?) Quali miglioramenti si potrebbero apportare al programma, tenendo condo di questo fatto?

9-5. L'uso di una tabella di salto ordinata

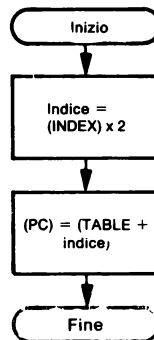
Scopo: Usare il contenuto della variabile INDEX, alla locazione 6000, come indice per una tabella che inizia a TABLE (locazione 6002). Ogni elemento della tabella contiene un indirizzo a 16 bit. Il programma deve trasferire il controllo all'indirizzo corrispondente all'indice: cioè, se l'indice è 6,

il programma salta all'indirizzo che occupa la posizione numero 6 della tabella. (Abbiamo iniziato a contare partendo dalla posizione 0.)

Problema Campione:

```
INDEX     - (6000) = 0002
TABLE     - (6002) = 4740    Elemento 0-esimo
            (6004) = 47A6
            (6006) = 47D0
            (6008) = 4620
            (600A) = 4854    Quarto elemento
```

Risultato: (PC) = 0047D0, in quanto questo è il secondo elemento della tabella (partendo da zero). L'istruzione successiva da eseguire è quella posta a questo indirizzo.

Diagramma di Flusso 9-5

L'ultimo blocco del diagramma ha come conseguenza il trasferimento del controllo all'indirizzo prelevato dalla tabella, per cui non è necessario un blocco che ne indichi la fine. Questo non crea nessun problema al processore, ma è opportuno che inseriate alcune note nel vostro diagramma di flusso e nella documentazione del programma, in modo che la sequenza non appaia come un vicolo cieco.

Programma 9-5a

```
00006000:                  DATA      EQU      $6000
00004000:                  PROGRAM   EQU      $4000
                           ;
00006000:                  INDEX     EQU      $6000          INDICE NELLA TABELLA
00006002:                  TABLE     EQU      $6002          INIZIO DELLA TABELLA
                           ;
                                     ORG      PROGRAM

00004000: 3078 6000        PGM_9_5A  MOVEA.W  INDEX,A0       PRENDI INDICE TABELLA
00004004: D0C8                       ADDA.W   A0,A0          CORREGGI INDICE PER OFFSET WORD
00004006: 3268 6002                  MOVEA.W  TABLE(A0),A1   PRENDI IND. DALLA TABELLA
0000400A: 4ED1                       JMP      (A1)           VAI A QUELL'INDIRIZZO

                                     END      PGM_9_5A
```

Quando eseguite questo programma, assicuratevi di aver messo del codice eseguibile (ad es. un'istruzione TRAP) ad ognuno degli indirizzi, ai quali può essere trasferito il controllo. Altrimenti, il processore eseguirà del codice privo di senso e non sarà possibile stabilire dove è avvenuta la diramazione.

Tabelle di salto

Le tabelle di salto si rivelano molto utili quando il processore deve scegliere quale routine eseguire fra quelle disponibili. Queste situazioni si verificano durante la decodifica di comandi (immessi, ad esempio, tramite una tastiera di controllo), nella selezione di procedure di collaudo, nella scelta di metodi o unità alternative o nella selezione di una particolare configurazione di I/O. Ad esempio, un interruttore a quattro posizioni in un'apparecchiatura di collaudo permette di selezionare altrettanti modi operativi: comandato a distanza, automatico, manuale o con auto-verifica. Il processore legge l'interruttore e seleziona la routine corrispondente servendosi di una tabella di salto. I volumi indicati ai punti 10 e 11 della Bibliografia descrivono altre possibili utilizzazioni.

La tabella di salto sostituisce un'intera serie di operazioni di salto condizionato, rendendo un programma compatto, efficiente e facilmente modificabile.

L'indice che guida l'accesso alla tabella deve essere moltiplicato per 2 per avere uno spostamento pari ad una word, poichè ciascun elemento è un indirizzo a 16 bit che occupa due byte di memoria. Questo è dovuto al fatto che gli indirizzi contenuti nella tabella sono di tipo assoluto corto. Su quale presupposto si basa, inoltre, il programma a proposito della lunghezza della tabella di salto?

Se gli indirizzi dovessero indicare una qualunque delle locazioni dei 16 megabyte di memoria potenzialmente indirizzabili dal processore, allora ciascun elemento richiederebbe almento tre byte. Nel caso di elementi di cinque byte, si potrebbe risolvere il problema semplicemente inserendo un'ulteriore istruzione ADDA e trasformando l'istruzione MOVE.W TABLE(A0),A1 in una MOVE.L. Si incontreranno, tuttavia, delle difficoltà, cercando di mettere la tabella di salto del Progamma 9-5a a indirizzi maggiori di 7FFF. Perche?

Il Programma 9-5b indica come utilizzare una tabella di salto, avvalendosi dell'indirizzamento indicizzato.

Listato Programma 9-5b

```
00006000:                DATA      EQU      $6000
00004000:                PROGRAM   EQU      $4000
                         ;
00006000:                INDEX     EQU      $6000              INDICE NELLA TABELLA DI SALTO
00006002:                TABLE     EQU      $6002              INIZIO DELLA TABELLA  DI SALTO
                         ;
                                   ORG      PROGRAM
00004000: 207C 0000
00004004: 6002           PGM_9_5B  MOVEA.L  #TABLE,A0          PRENDI INDIRIZZO TABELLA
00004006: 3038 6000                MOVE.W   INDEX,D0           PRENDI INDICE TABELLA
0000400A: E540                     ASL.W    #2,D0              CORREZ. PER ELEMENTI DA 4 BYTE
0000400C: 2270 0000                MOVEA.L  0(A0,D0.W),A1      PRENDI IND. DALLA TAB. DI SALTO
00004010: 4ED1                     JMP      (A1)               VAI A QUELL'INDIRIZZO

                                   END      PGM_9_5B
```

Livelli di "indirettezza"

In entrambi questi programmi, l'istruzione JMP (A1) è un salto indiretto che trasferisce il contenuto del registro A1 nel contatore di programma. Questa istruzione può, in certi casi, confondere le idee, per la difficoltà di stabilire fino a che punto sia "indiretta". Per chiarire questo concetto, confrontiamo come agisce JMP (A1) rispetto a MOVEA (A1),A0 Nel caso di JMP (A1) il contatore di programma riceve il valore contenuto in A1, mentre con l'istruzione MOVEA (A1),A0 il registro A0 riceve il valore presente nella locazione indicata in A1.

Questa è una apparente incoerenza nella sintassi del linguaggio assembly. Può essere superata leggendo l'istruzione JMP (A1) in questo modo:

"Salta alla locazione indicata da A1."

Cosa sarebbe accaduto se avessimo sostituito le ultime due istruzioni del Programma 9-5b con JMP TABLE(A0)?

Come si potrebbe modificare il Programma 9-5b in modo che preveda, nella variabile TABLE, l'indirizzo della tabella, anzichè l'inizio della tabella stessa?

PROBLEMI

9-1. Rimuovere un elemento da una lista

Scopo: Rimuovere il valore contenuto nella variabile ITEM, alla locazione di memoria 6000, da una lista qualora sia presente in essa. L'indirizzo della lista si trova nella variabile LIST, alla locazione 6002. Il primo elemento della lista indica il numero (in word) dei restanti elementi. Spostare di una posizione gli elementi successivi a quello rimosso e ridurre di 1 la lunghezza della lista.

Problemi Campione:

a. Input:	ITEM	– (6000)	=	D010	Elemento da rimuovere
	LIST	– (6002)	=	00005000	Ind. della lista
		(5000)	=	0004	Lungh. della lista
		(5002)	=	C121	Primo elemento
		(5004)	=	A346	
		(5006)	=	3A64	
		(5008)	=	6C20	

Risultato: Nessun cambiamento, poichè l'elemento da rimuovere non è nella lista.

b. Input:	ITEM	– (6000)	=	D010	Elemento da rimuovere

	LIST	– (6002)	=	00005000	Ind. della lista
		(5000)	=	0004	Lungh. della lista
		(5002)	=	C121	Primo elemento
		(5004)	=	D010	
		(5006)	=	3A64	
		(5008)	=	6C20	
Risultato:		(5000)	=	0003	Lungh. della lista
		(5002)	=	C121	diminuita di 1
		(5004)	=	3A64	Gli altri elementi
		(5006)	=	6C20	spostati di una posizione

9-2. Aggiunta di un elemento ad una lista ordinata

Scopo: Inserire il valore contenuto nella variabile ITEM, alla locazione 6000, in una lista ordinata qualora non sia già presente. L'indirizzo della lista è nella variabile LIST, alla locazione 6002. Il primo elemento della lista indica la sua lunghezza in word. La lista vera e propria consiste di numeri binari privi di segno, in ordine crescente. Mettere il nuovo ingresso nella posizione corretta, spostando gli elementi successivi ed aumentando di 1 la lunghezza della lista.

Problemi Campione:

a. Input:	ITEM	– (6000)	=	7010	Elem. da aggiungere
	LIST	– (6002)	=	00005000	Ind. della lista
		(5000)	=	0004	Lungh. della lista
		(5002)	=	0037	Primo elemento
		(5004)	=	5322	
		(5006)	=	A101	
		(5008)	=	C203	
Risultato:		(5000)	=	0005	Lungh. della lista
		(5002)	=	0037	aumentata di 1
		(5004)	=	7010	Nuovo elemento
		(5006)	=	A101	Altri elem. spostati
		(500A)	=	C203	di una posizione
b. Input:	ITEM	– (6000)	=	7010	Elem. da aggiungere
	LIST	– (6002)	=	00005000	Ind. della lista
		(5000)	=	0004	Lungh. della lista
		(5002)	=	0037	Primo elemento
		(5004)	=	5322	
		(5006)	=	7010	
		(5008)	=	C203	

Risultato:Nessun cambiamento, perchè il valore è già in lista.

9-3. Aggiunta di un elemento ad una coda

Scopo: Aggiungere il valore presente nella variabile ITEM, alla locazione di memoria 6000, ad una coda. L'indirizzo del primo elemento della coda è nella variabile QUEUE, alla locazione 6002. Ogni elemento della coda contiene l'indirizzo dell'elemento successivo oppure zero se un elemento successivo non esiste. Il nuovo elemento viene messo in fondo alla coda; il suo indirizzo verrà a trovarsi nell'elemento che era all'estremità della coda. Il nuovo elemento conterrà zero per indicare la fine della coda.

Problema Campione:

Input:	ITEM	– (6000)	= 000060A0	
	QUEUE	– (6002)	= 00006020	Punt. inizio coda
		(6020)	= 00006030	
		(6030)	= 0000	Ultimo elemento nella coda
Risult.:	QUEUE	– (6002)	= 00006020	
		(6020)	= 00006030	
		(6030)	= 000060A0	L'ultimo vecchio elem. punta al nuovo
		(60A0)	= 0000	Nuovo elemento finale

Come sarebbe possibile aggiungere un nuovo elemento alla coda se alla locazione 6006 fosse contenuto l'indirizzo dell'ultimo elemento? (Ricordarsi di aggiornare questo puntatore alla fine della coda).

9-4. Ordinamento di valori a 32 bit

Scopo: Ordinare una lista di elementi di 4 byte in senso decrescente. I primi tre byte di ciascun elemento costituiscono una chiave senza segno, con il byte più significativo per primo. Il quarto byte è un'ulteriore informazione che non deve essere usata in fase di ordinamento, ma solamente spostata insieme alla relativa chiave. Il numero di elementi presenti nella lista è indicato dalla variabile word LENGTH, alla locazione 6000. La lista vera e propria inizia alla locazione 6002 (LIST).

Problema Campione:

Input:	LENGTH	– (6000)	= 0004	4 elem. nella lista
	LIST	– (6002)	= 41	Iniz.chiave primo el.
		(6003)	= 42	
		(6004)	= 43	Fine chiave primo el.
		(6005)	= 07	Informazione addiz. del primo elemento
		(6006)	= 4A	Secondo elemento
		(6007)	= 4B	
		(6008)	= 4C	
		(6009)	= 13	
		(600A)	= 4A	Terzo elemento
		(600B)	= 4B	
		(600C)	= 41	
		(600D)	= 37	
		(600E)	= 44	Quarto elemento
		(600F)	= 4B	
		(6010)	= 41	
		(6011)	= 3F	
Risult.:	LIST	– (6002)	= 4A	
		(6003)	= 4B	
		(6004)	= 4C	
		(6005)	= 13	Fine primo elemento.
		(6006)	= 4A	
		(6007)	= 4B	
		(6008)	= 41	
		(6009)	= 37	Fine secondo elemento
		(600A)	= 44	
		(600B)	= 4B	
		(600C)	= 41	
		(600D)	= 3F	Fine terzo elemento
		(600E)	= 41	
		(600F)	= 42	
		(6010)	= 43	
		(6011)	= 07	Fine ultimo elemento

I dati degli elementi non ordinati sono 'ABC',$07; 'JKL',$13; 'JKA',$37; 'DKA',$3F.

9-5. Utilizzazione di una tabella di salto con una chiave

Scopo: Usare il valore presente nella variabile INDEX, alla locazione di memoria 6000, come chiave per una tabella di salto (TABLE), che inizia alla locazione 6002. Ogni elemento della tabella contiene un valore a 16 bit, seguito da un indirizzo a 32 bit, al quale il programma deve trasferire il controllo, se la chiave è uguale a quel determinato valore.

Problema Campione:

Input:	INDEX	– (6000)	= 4142	
	TABLE	– (6002)	= 4348	Primo valore chiave
		(6004)	=	00004900
				Primo indirizzo
		(6008)	= 4142	Secondo valore chiave
		(600A)	=	00004940
		(600E)	= 4558	Terzo valore
		(6010)	=	00004A2
				0

Risult.: (PC) - 004940, poichè questo indirizzo corrisponde al valore chiave 4142

BIBLIOGRAFIA

1 J. Hemenway and E. Teja. "EDN Software Tutorial: Hash Coding," *EDN*, September 20, 1979, pp. 108-10.

2 D. Knuth. *The Art of Computer Programming, Volume III: Searching and Sorting*. Reading Mass.: Addison-Wesley, 1978.

3 D. Knuth. "Algorithms," *Scientific American,* April 1977, pp. 63-80.

4 K.J. Thurber and P.C. Patton. *Data Structures and Computer Architecture*. Lexington Mass.: Lexington Books, 1977.

5 J. Hemenway and E. Teja. "Data Structures - Part 1," *EDN*, March 5, 1979, pp. 89-92; "Data Structures - Part 2," *EDN*, May 5, 1979, pp. 113-16.

6 See Reference 2.

7 B.W. Kernighan and P.J. Plauger. *The Elements of Programming Style*. New York: McGraw-Hill, 1978.

8 K.A. Schember and J.R. Rumsey. "Minimal Storage Sorting and Searching Techniques for RAM Applications," *Computer*, June 1977, pp. 92-100.

9 "Sorting 30 Times Faster with DPS," *Datamation,* February 1978, pp. 200-03.

10 L.A. Leventhal. "Cut Your Processor's Computation Time," *Electronic Design,* August 16, 1977, pp. 82-89.

11 J.B. Peatman. *Microcomputer-Based Design*. New York: McGraw-Hill, 1977, Chapter 7.

SEZIONE III

ARGOMENTI PIÙ COMPLESSI

I capitoli seguenti tratteranno aspetti più complessi della programmazione in linguaggio assembly. I Capitoli 10 e 11 affronteranno l'argomento delle subroutine, una componente importante di tutti i livelli di programmazione. Il Capitolo 10 prende in esame le tecniche per il passaggio dei parametri, mentre il Capitolo 11 definisce le subroutine, mostrando degli esempi. I tre capitoli successivi descrivono le modalita di input/ouput, che rappresentano il modo in cui un microprocessore comunica con il mondo esterno. Nel Capitolo 12 sono esaminate le routine di ritardo e i differenti tipi di periferiche. Il Capitolo 13 si occupa del PIA 6821 (Parallel Interface Adapter), un dispositivo parallelo di I/O destinato ai processori della Motorola, e fornisce alcuni esempi di programmi in grado di gestirlo. Il Capitolo 14 illustra le routine fondamentali per la gestione di un dispositivo di interfacciamento seriale, l'ACIA 6850 (Asynchronous Communications Interface Adapter). Il Capitolo 15 tratta l'importante e complesso argomento degli interrupt ed altri tipi di trattamento delle Exception, tipici dell'MC68000.

CAPITOLO 10

TECNICHE PER IL PASSAGGIO DI PARAMETRI

Nessuno degli esempi che vi abbiamo mostrato finora può esser considerato un programma vero e proprio. La maggior parte dei programmi applicativi svolge tutta una serie di funzioni, molte delle quali sono richiamate più volte e, in certi casi, utilizzate anche da altri programmi.

SUBROUTINE

Il metodo impiegato per realizzare programmi che abbiano queste caratteristiche consiste nello scrivere delle subroutine destinate a svolgere compiti particolari. Le sequenze di istruzioni che ne fanno parte sarà sufficiente scriverle e collaudarle solo una volta utilizzandole poi ripetutamente.

Una subroutine per poter essere veramente utile deve essere generica. Ad esempio, una subroutine in grado di svolgere soltanto una funzione molto particolare, come cercare una determinata lettera in una stringa di lunghezza costante, finirà per non essere di grande utilità. Se, invece, quella subroutine è capace di ricercare una lettera qualsiasi in una stringa di lunghezza variabile, allora sarà senz'altro utilizzabile in un maggior numero di casi.

Per garantire una flessibilità di questo tipo è necessario che le subroutine abbiano la capacità di ricevere vari tipi di informazioni, rappresentate sostanzialmente da dati o indirizzi e che indichiamo con il termine di *parametri*. Un aspetto importante di cui bisogna tener conto nella stesura di una subroutine è appunto il trasferimento dei relativi parametri. Questo procedimento lo definiamo Passaggio di Parametri.

TECNICHE GENERALI PER IL PASSAGGIO DI PARAMETRI

Esistono tre metodi fondamentali per trasferire dei parametri:

1. Mettere i parametri nei registri.

2. Mettere i parametri immediatamente dopo la chiamata della subroutine, nella memoria di programma.
3 Trasferire i parametri ed i risultati utilizzando lo stack.

I registri rappresentano spesso un mezzo rapido ed efficace per trasferire dei parametri e contenere, poi, i risultati dell'elaborazione. L'unica limitazione è costituita dal numero dei registri disponibili; inoltre, questo metodo causa spesso degli effetti collaterali imprevedibili e non è sufficientemente generale.

Bisogna trovare un compromesso fra un tempo di esecuzione abbastanza breve ed una soluzione che sia valida nel maggior numero possibile di situazioni. Un compromesso di questo tipo è molto comune nelle applicazioni degli elaboratori, a tutti i livelli. Una soluzione di carattere generale è anche più facile da imparare e, inoltre, il suo impiego può essere automatizzato mediante l'uso delle macro. D'altra parte, subroutine destinate a svolgere solo una particolare funzione sono molto più veloci e richiedono una minore disponibilità di memoria. La scelta dipende dal tipo di applicazione, ma è sempre preferibile adottare una soluzione più generale (risparmiando tempo in fase di programmazione e semplificando la documentazione), tranne in quei casi in cui limitazioni di tempo e di memoria non costringano a fare diversamente.

Passaggio di Parametri tramite i Registri

Il metodo più semplice per trasferire dei parametri ad una subroutine è, appunto, attraverso i registri. Prima di chiamare una subroutine il programma principale deve caricare nei registri i valori relativi a indirizzi, contatori ed eventuali altri dati. Ad esempio, supponiamo che una subroutine agisca su due buffer di dati della stessa lunghezza. Essa richiede che la lunghezza dei buffer sia presente nel registro D0, mentre gli indirizzi iniziali dei buffer devono trovarsi nei registri A0 e A1. Il programma principale richiamerà, allora, la subroutine nel modo seguente:

```
MOVE.W     #BUFL,DO     LUNGHEZZA DEL BUFFER IN DO
MOVEA.L    BUFA,A0      INDIRIZZO INIZIALE DEL BUFFER A IN A0
MOVEA.L    BUFB,A1      INDIRIZZO INIZIALE DEL BUFFER B IN A1
JSR        SUBR         CHIAMATA DELLA SUBROUTINE
```

Con questo metodo una subroutine si aspetta semplicemente di trovare i parametri in quei determinati registri. Riguardo ai risultati dell'elaborazione possiamo metterli in alcuni dei registri oppure fornire ad una subroutine, sempre mediante i registri, gli indirizzi delle locazioni destinate alla loro memorizzazione. Naturalmente, la limitazione di questa tecnica sta nel numero dei registri disponibili. Alcune caratteristiche dell'MC68000, come l'indirizzamento indiretto a registri, la possibilità di usare un qualsiasi registro indirizzi

come puntatore dello stack e l'istruzione LEA, costituiscono mezzi molto più potenti e generali per il trasferimento di parametri.

Passaggio di Parametri nella Memoria di Programma

I parametri possono trovarsi immediatamente dopo la chiamata della subroutine, che, in tal caso, oltre a prelevare i parametri, dovrà anche modificare l'indirizzo di ritorno alla sommità dello stack. Adottando questa tecnica dovremo apportare le seguenti modifiche al nostro esempio:

```
JSR     SUBR
DC.W    BUFL    LUNGHEZZA DEL BUFFER
DC.L    BUFA    INDIRIZZO INIZIALE DEL BUFFER A
DC.L    BUFB    INDIRIZZO INIZIALE DEL BUFFER B
```

La subroutine salva il contenuto dei registri della CPU e, quindi, carica i parametri, modificando l'indirizzo di ritorno come segue:

```
SUBR    MOVEM.L    D0/A0-A2,-(A7)    LA SUBROUTINE USA
                                     D0,A0,A1,A2
        MOVEA.L    16(A7),A2         L'IND. DI RITORNO PUNTA A
                                     BUFL
        MOVE.W     (A2)+,D0          BUFL IN DO
        MOVEA.L    (A2)+,AO          BUFA IN A0
        MOVEA.L    (A2)+,A1          BUFB IN A1
        MOVEA.L    A2,16(A7)         AGGIUSTA L'INDIRIZZO DI
                                     RITORNO
```

La costante 16 serve per ovviare al cambiamento avvenuto in A7 quando i quattro registri D0, A0, A1, e A2 sono stati salvati sullo stack.

Questa tecnica per il passaggio dei parametri ha il vantaggio di essere facilmente comprensibile; tuttavia, richiede che i parametri siano definiti nel momento in cui viene scritto il programma. Allo scopo di rendere possibile una successiva variazione dei parametri, possiamo far seguire la chiamata della subroutine da un puntatore che indichi l'area di memoria dove si trovano i parametri veri e propri. Ecco un esempio:

```
        JSR     SUBR
        DC.L    PLIST      INDIRIZZO DI INIZIO DEI
                           PARAM.

PLIST   DC.W    WBUFL
        DC.L    BUFA
        DC.L    BUFB
```

```
SUBR    MOVEM.L   D0/A0-A2,-(A7)    LA SUBR. USA D0,A0,A1,
                                    A2
        MOVEA.L   16(A7),A1         L'IND. DI RITORNO PUNTA
                                    A PLIST
        MOVEA.L   (A1)+,A2          PRENDI L'IND. DELLA LI-
                                    STA PARAM.
        MOVEA.L   A1,16(A7)         ...E AGGIORNA L'IND. DI RI-
                                    TORNO
        MOVE.L    (A2)+,D0          BUFL IN DO
        MOVEA.L   (A2)+,A0          BUFA IN A0
        MOVEA.L   (A2)+,A1          BUFB IN A1
```

L'insieme dei parametri contenuto in un'area separata della memoria è indicato come "blocco parametri". Nell'esempio precedente dopo JSR abbiamo messo l'indirizzo iniziale di un blocco, formato da tre word. Questo indirizzo potevamo fornirlo alla subroutine anche in un altro modo:

```
MOVE.L  #PLIST,-(A7)  METTI L'IND. DEL BLOCCO PARAMETRI
        SULLO STACK
JSR     SUBR
```

A sua volta, essa avrebbe prelevato i parametri nel modo seguente:

```
SUBR    MOVEM.L   D0/A0-A2,-(A7)    LA SUBROUTINE USA D0, A0,
                                    A1, A2
        MOVEA.L   20(A7),A2         PRENDI L'INDIRIZZO DEI PA-
                                    RAMETRI
        MOVE.W    (A2)+,D0          BUFL IN DO
        MOVEA.L   (A2)+,A0          BUFA IN A0
        MOVEA.L   (A2)+,A1          BUFB IN A1
```

Quando si impiega questo metodo non è necessario aggiustare il puntatore allo stack.

Affinchè il programma principale possa disporre dei risultati, essi saranno memorizzati nello stesso blocco parametri oppure in altre locazioni, i cui indirizzi saranno anch'essi forniti alla subroutine come parametri.

Passaggio di Parametri sullo Stack

Un altro metodo piuttosto comune per trasferire dei parametri ad una subroutine prevede l'uso dello stack. Adottando questa tecnica la chiamata della subroutine illustrata in precedenza avverrebbe in questo modo:

```
        MOVE.W    #BUFL,-(A7)        METTI SULLO STACK LA
                                     LUNGH. DEL BUFFER
        MOVEA.L   BUFA,-(A7)         METTI GLI INDIRIZZI DEI
                                     BUFFER
        MOVEA.L   BUFB,-(A7)         ... SULLO STACK
        JSR       SUBR
```

All'inizio la subroutine dovrà caricare i parametri nei registri della CPU:

```
SUBR    MOVEM.L   D0/A0/A1,-(A7)     SALVA IL CONTENUTO DEI
                                     REGISTRI
        MOVEA.L   12(A7),A1          INDIRIZZO INIZIALE DEL
                                     BUFFER B IN A1
        MOVEA.L   16(A7),A0          INDIRIZZO INIZIALE DEL
                                     BUFFER A IN A0
        MOVE.W    20(A7),D0          LUNGHEZZA DEL BUFFER IN
                                     D0
```

Con una soluzione di questo tipo, sia il passaggio dei parametri che il trasferimento dei risultati avviene sullo stack.

Lo stack dell'MC68000 si accresce verso il basso (verso gli indirizzi più bassi della memoria). Questo è dovuto al fatto che i vari elementi sono messi sullo stack (operazione di push) tramite l'indirizzamento con predecremento, per cui il puntatore dello stack contiene sempre l'indirizzo dell'ultima locazione occupata, anzichè quello della successiva locazione disponibile, come avviene in altri microprocessori (ad es. il 6800). Questo spiega la necessità di inizializzare il puntatore con un valore uperiore a quello dell'indirizzo più alto dell'area di memoria utilizzata come stack.

Se vuole trasferire dei parametri tramite lo stack, un programmatore deve seguire questa procedura:

1. Decrementare il puntatore dello stack di sistema per creare uno spazio, dove mettere i parametri utilizzando degli offset rispetto al puntatore di stack; o, più semplicemente, mettere direttamente i parametri sullo stack.

2. Accedere ai parametri mediante degli offset rispetto al puntatore dello stack di sistema, ricordando che JSR mette alla sommità dello stack l'indirizzo di ritorno.

3. Salvare i risultati sullo stack, sempre utilizzando degli offset rispetto al puntatore dello stack di sistema.

4. Ripulire lo stack prima o dopo il ritorno dalla subroutine, in modo da rimuovere i parametri e poter disporre dei risultati.

TIPI DI PARAMETRI

Indipendentemente dalla tecnica adottata per il loro passaggio, i parametri possono essere specificati in vari modi. Ad esempio:

1. Mettere i valori dei parametri in una lista. Questo metodo è conosciuto anche come "chiamata per valore", dal momento che dobbiamo preoccuparci solo dei valori dei parametri.

2. Mettere nella lista gli indirizzi dei parametri. Questo metodo è definito anche "chiamata per nome", poichè dobbiamo indicare le locazioni occupate dai parametri. (Più precisamente questo metodo è definito "chiamata per reference", in quanto con il termine "chiamata per nome" ci si riferisce usualmente ad un'altra tecnica più complessa utilizzata in alcuni linguaggi ad alto livello come l'ALGOL60 e il SIMULA67.).

CAPITOLO 11

SUBROUTINE

Istruzioni di chiamata di una subroutine e di ritorno al programma principale

La maggior parte dei microprocessori dispone di istruzioni speciali per trasferire il controllo ad una subroutine e ritornare successivamente al programma principale. Del primo gruppo fanno parte istruzioni come Call, Jump-to-Subroutine, Jump-and-Mark Place o Jump-and-Link. L'istruzione che serve a restituire il controllo al programma principale è, di solito, indicata con Return.

Sul microprocessore MC68000, le istruzioni Jump-to-Subroutine (JSR) o Branch-to-Subroutine (BSR) salvano il valore del contatore di programma sullo stack prima di sostituirlo con l'indirizzo iniziale della subroutine; l'istruzione Return-from-Subroutine (RTS) prende il vecchio valore dallo stack e lo rimette nel contatore di programma. L'effetto ottenuto è quello di trasferire il controllo, prima alla subroutine e poi, di nuovo, al programma principale. Chiaramente una subroutine può, a sua volta, richiamare un'altra subroutine, e così via.

TIPI DI SUBROUTINE

Subroutines Rilocabili

Alcune volte le subroutine devono possedere delle particolari caratteristiche. **Una subroutine è rilocabile, se può essere collocata in una zona qualsiasi della memoria.** Questo ci consente di utilizzarla, senza tener conto della eventuale presenza di altri programmi o del modo in cui è organizzata la memoria. **Con una routine di questo tipo è necessario un caricatore rilocante, che provvederà a metterla in memoria dopo le altre routine ed aggiungerà il suo indirizzo iniziale, o costante di rilocazione, a tutti gli indirizzi presenti nella subroutine. Un codice indipendente dalla posizione non ha bisogno di un simile caricatore, in quanto tutti gli indirizzi sono espressi, relativamente al valore del contatore di programma, mentre gli indirizzi dei dati sono contenuti sempre nei registri.** In una parte successiva di questo capitolo, descriveremo le modalità per la realizzazione di un codice indipendente dalla sua posizione.

Subroutines Rientranti

Una subroutine viene detta rientrante, se può essere interrotta e richiamata dal programma che ha causato l'interruzione, fornendo ancora risultati corretti ad entrambi i programmi, l'interrotto e l'interrompente. Questa è una caratteristica importante in un sistema basato sugli interrupt; altrimenti, le routine che servono gli interrupt non possono utilizzare le subroutine standard, senza causare degli

errori. Le subroutine di un microprocessore si prestano facilmente ad essere rientranti, dal momento che l'istruzione Call usa lo stack e questa è una procedura automaticamente rientrante. Il solo requisito da rispettare è che la subroutine utilizzi soltanto i registri e lo stack, e non delle locazioni fisse, per la memorizzazione temporanea di dati.

Subroutines Ricorsive

Una subroutine è ricorsiva, se richiama se stessa. Una subroutine di questo tipo deve, chiaramente, essere anche rientrante.

DOCUMENTAZIONE DI UNA SUBROUTINE

La gran parte dei programmi è costituita da un programma principale e da varie subroutine. Questo è un vantaggio importante, perchè rende possibile l'impiego di routine già esistenti e collaudate e, quindi, ci consente di dedicarci interamente alla correzione ed al collaudo delle subroutine di nuova realizzazione, delle quali vanno analizzati, con attenzione, gli effetti sui registri e le locazioni di memoria.

Il listato di una subroutine deve fornire una quantità di informazioni tale da non costringere gli eventuali utenti ad esaminare la sua struttura interna. Ecco alcune delle cose da specificare:

- Una descrizione della funzione della subroutine
- Una lista dei parametri in ingresso ed in uscita
- I registri e le locazioni di memoria utilizzate
- Un esempio del suo impiego, includendo, eventualmente, una tipica sequenza di richiamo

Seguendo queste indicazioni, l'impiego di una subroutine risulterà estremamente semplice.

ESEMPI DI PROGRAMMAZIONE

Gli esempi contenuti in questo capitolo presuppongono che lo stack ed il relativo puntatore siano già stati inizializzati. Non abbiamo, infatti, indicato quelle istruzioni che caricano un indirizzo nel puntatore allo stack o provvedono ad azzerare quest'ultimo, prima di utilizzarlo. Se desiderate definire una vostra area di stack, ricordate di salvare il puntatore esistente e di ripristinarlo successivamente, per garantire un corretto ritorno alla fine del programma. Dato che l'MC68000 permette di usare come stack un registro indirizzi qualsiasi, è preferibile servirsi di un proprio stack, senza cambiare il puntatore allo stack di sistema (A7).

Dal momento che l'MC68000 non dispone di speciali istruzioni per caricare o salvare il valore allo stack, dovremo servirci dell'istruzione MOVEA per modificare il registro di stack, come abbiamo fatto nel programma seguente.

Listato Programma 11-0

```
00006000:                   DATA     EQU      $6000
00004000:                   PROGRAM  EQU      $4000
                            ;
00006000:                   PSTACK   EQU      DATA
00008000:                   STACK    EQU      $8000
                            ;
00004600:                   MAIN     EQU      $4600
                            ;
                                     ORG      PROGRAM

00004000: 21CF 6000                  MOVEA.L A7,PSTACK      SALVA STACK PRECEDENTE
00004004: 2E7C 0000
00004008: 8000                       MOVEA.L #STACK,A7      PREDISPONI NUOVO STACK
0000400A: 4EB8 4600                  JSR     MAIN
0000400E: 2E78 6000                  MOVEA.L PSTACK,A7      RIPRISTINA STACK PRECEDENTE
                            ;
00004012: 4E75                       RTS

                                     END
```

Spiegazione del programma 11-0

Questa routine salva il puntatore allo stack, ne crea uno nuovo e, infine, richiama il programma MAIN, che disporrà, così, di uno stack che inizia a 8000. Alla fine del programma dovrà esserci un'istruzione RTS, che trasferisca il controllo a questa routine, la quale, a sua volta, provvederà a ripristinare il vecchio puntatore, garantendo un ritorno corretto al programma precedente.

ESEMPI DI PROGRAMMAZIONE

11-1. CONVERSIONE DA ESADECIMALE AD ASCII

Scopo: Convertire il contenuto del registro dati D0 nel corrispondente carattere ASCII. Si presuppone che il contenuto originale del registro dati D0 sia minore di 16.

Problemi Campione:

a.	Input:	D0 = 0C
	Risultato:	D0 = 43 'C'
b.	Input:	D0 = 06
	Risultato:	D0 = 36 '6'

Effetti dell'esecuzione dell'istruzione JSR

L'istruzione JSR salva il valore presente nel contatore di programma (che è poi l'indirizzo dell'istruzione successiva a JSR) sullo stack di sistema e lo sostituisce con l'indirizzo della subroutine. La procedura è questa:

Fase 1. Decrementa di 4 il puntatore allo stack
Fase 2. Salva il contatore di programma nella word posta alla sommità dello stack.
Fase 3 Mette l'indirizzo iniziale della subroutine nel contatore di programma.

Ecco cosa accade, nel caso del Programma 11-1, quando viene eseguita l'istruzione JSR:

```
Prima di JSR
        PC = 004604
        A7 = 7FFC

Dopo l'esecuzione di JSR
        PC  = 00460E
        A7  = 7FF8
        (7FF8) = 00004608
```

Il puntatore allo stack viene sempre decrementato di quattro, poichè tutti gli indirizzi sono memorizzati sullo stack come valori a 32 bit, anche se con gli indirizzi di ritorno è consentito l'uso del modo assoluto corto. Quando il processore preleva l'istruzione JSR, incrementa anche il contatore di programma, che indica così l'istruzione immediatamente successiva a JSR: è questo l'indirizzo salvato sullo stack come valore a 32 bit.

L'istruzione BSR

L'istruzione JSR è simile all'istruzione JMP, tranne per il fatto di "ricordare" da dove proviene. A questo proposito, va sottolineato che JSR può richiamare una subroutine posta in un'area qualsiasi della memoria. JSR è correlata all'istruzione BSR, come JMP lo è con BRA. Anche BSR serve a richiamare una subroutine ed a mettere allo stack l'indirizzo di ritorno, ma, come accade per BRA, è utilizzabile solamente quando l'istruzione a cui trasparire il controllo si trova in un'area di memoria raggiungibile con un valore di spostamento di 8 o 16 bit.

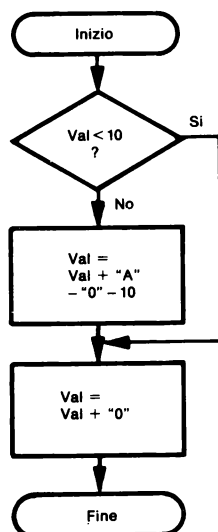
Effetto dell'esecuzione dell'istruzione RTS

L'istruzione RTS inverte il processo:

Fase 1 Mette il valore che si trova alla sommità allo stack nel contatore di programma.
Fase 2. Incrementa di 4 il puntatore allo stack.

Ecco cosa accade con l'istruzione RTS, nel Programma 11-1:

```
Prima di RTS
        PC  = 00461A
        A7  = 7FF8
        (7FF8) = 00004608

Dopo l'esecuzione di RTS
        PC  = 00468
        A7  = 7FFC
```

Diagramma di Flusso 11-1

Il programma principale prende il dato dalla variabile HDIGIT, alla locazione di memoria 6000, chiama la subroutine di conversione e salva il risultato nella variabile ACHAR, alla locazione 6001.

Programma 11-1

```
00006000:                     DATA     EQU     $6000
00004600:                     PROGRAM  EQU     $4600
                              ;
                                       ORG     DATA

00006000:                     HDIGIT   DS.B    1            CIFRA ESADECIMALE DA CONVERTIRE
00006001:                     ACHAR    DS.B    1            CARATTERE ASCII CONVERTITO
                              ;
                                       ORG     PROGRAM

00004600: 1038 6000           MAIN     MOVE.B  HDIGIT,D0    PRENDI DATO:VARIA DA 00 A 0F 0F
00004604: 4EB9 0000
00004608: 4610                         JSR     HEXDIGIT     CONVERTI AD UN CARATT. ASCII
0000460A: 11C0 6001                    MOVE.B  D0,ACHAR
0000460E: 4E75                         RTS
                              *
                              * SUBROUTINE HEXDIGIT
                              *
                              * SCOPO: HEXDIGIT CONVERTE UNA CIFRA ESAD. IN UN CARATTERE ASCII
                              *
                              * CONDIZIONI INIZIALI: D0.B CONTIENE UN VALORE DA 00 A 0F 0F
                              *
                              * CONDIZIONI FINALI:    D0.B CONTIENE UN CARATTERE ASCII COMPRESO
                              *                       FRA '0'-'9' O 'A'-'F'
                              *
                              * REGISTRI CAMBIATI:    SOLO D0
                              *
                              * CASO CAMPIONE:        CONDIZIONI INIZIALI: D0.B = 6
                              *                       CONDIZIONI FINALI:   D0.B = 36 ('6')
                              *
                              *
00004610: 0C00 000A           HEXDIGIT CMP.B   #$0A,D0      CIFRA DECIMALE O LETTERA ESADEC.?
00004614: 6D02                         BLT.S   ADDZ         SE E' UNA CIFRA VAI A ADDZ
                              ;
00004616: 5E00                         ADD.B   #'A'-'0'-$0A,D0 OFFSET PER LE LETTERE
00004618: 0600 0030           ADDZ     ADD.B   #'0',D0        CONVERTI IN ASCII
                              ;
0000461C: 4E75                         RTS

                                       END     HEXDIGIT
```

L'MC68000 incrementa sempre il puntatore allo stack dopo aver prelevato un dato, per cui la procedura è analoga a quella dell'indirizzamento con postincremento. RTS bilancia gli effetti di JSR o di BSR, limitandosi, semplicemente, a prelevare quattro byte posti alla sommità allo stack e a metterli nel contatore di programma. Il programmatore deve accertarsi che essi contengano un indirizzo di ritorno valido, poichè il processore non si preoccupa minimamente di esaminarli.

Questa subroutine richiede un solo parametro e fornisce un unico risultato. Un registro dati è, evidentemente, il posto migliore dove mettere sia il parametro che il risultato.

Il programma principale svolge tre funzioni:

- Mette il dato nel registro dati.
- Chiama la subroutine.
- Salva il risultato.

Se preferite non utilizzare lo stack di sistema, ricordatevi della routine di inizializzazione descritta in precedenza, che permette appunto di definire un nuovo stack.

Questo è un programma rientrante, poichè non utilizza una memoria dati, ed è rilocabile, in quanto l'indirizzo ADDZ viene indicato relativamente al contatore di programma. L'impiego di BSR (Branch to Subroutine), al posto di JSR (Jump to Subroutine) renderebbe rilocabile anche il programma principale.

Tempo di esecuzione di JSR e di RTS

L'istruzione JSR equivale all'esecuzione di quattro o cinque istruzioni, occupando da 44 a 48 cicli di clock. Una chiamata di subroutine richiede un certo tempo, anche se avviene mediante una singola istruzione e quello del tempo è uno degli aspetti da non sottovalutare. Infatti, un'istruzione JSR richiede 10 cicli in più rispetto ad una istruzione JMP (con un identico tipo di indirizzamento), poichè deve salvare l'attuale valore del contatore di programma sullo stack. RTS occupa 16 cicli di clock.

11-2. CONVERSIONE DI UNA WORD ESADECIMALE IN UNA STRINGA ASCII

Scopo: Convertire il valore della variabile NUMBER, alla locazione di memoria 6000, in quattro cifre esadecimali, salvandole, poi, nel vettore di quattro byte STRING, che inizia alla locazione 6002. Eseguire questa conversione, facendo uso di una subroutine, che abbia come parametri il valore esadecimale e l'indirizzo della stringa.

Problema Campione:

```
Input:      NUMBER    (6000) = 4CD0
```

Risultato:	STRING	(6002) = 34 '4'
		(6003) = 43 'C'
		(6004) = 44 'D'
		(6005) = 30 '0'

Listato Programma 11-2

```
00006000:                       DATA     EQU      $6000
00004600:                       PROGRAM  EQU      $4600
                                ;
                                         ORG      DATA

00006000:                       NUMBER   DS.W     1                NUMERO DA CONVERTIRE IN ESADEC. ASCII
00006002:                       STRING   DS.B     4                STRINGA PER CIFRE ESADEC. ASCII
                                ;
                                         ORG      PROGRAM

00004600: 2F3C 0000
00004604: 6002                  MAIN     MOVE.L   #STRING,-(A7)    METTI IND. STRINGA SULLO STACK
00004606: 3F38 6000                      MOVE.W   NUMBER,-(A7)     SULLO STACK NUM. A 16 BIT DA CONV.
0000460A: 4EB9 0000
0000460E: 4612                           JSR      BINHEX           DA BINARIO AD ASCII/ESADECIMALE
                                ;
00004610: 4E75                           RTS
                                *
                                * SUBROUTINE BINHEX
                                *
                                * SCOPO: CONVERTIRE UN VALORE A 16 BIT IN 4 CIFRE ESAD. ASCII
                                *
                                * CONDIZIONI INIZIALI: IL PRIMO PARAMETRO SULLO STACK E' IL
                                *                      VALORE; IL SECONDO PARAMETRO E'
                                *                      L'INDIRIZZO DELLA STRINGA DA FORMARE
                                *
                                * CONDIZIONI FINALI:   LA STRINGA ESAD. OCCUPA 4 BYTE CONSECUTIVI
                                *                      A PARTIRE DALL'INDIRIZZO PASSATO COME
                                *                      SECONDO PARAMETRO
                                *
                                * USO DEI REGISTRI:    NESSUN REGISTRO E' MODIFICATO
                                *
                                * CASO CAMPIONE:       CONDIZIONI INIZIALI: 4CD0 IN CIMA ALLO STACK,
                                *                                           QUINDI 00006002
                                *                      CONDIZIONI FINALI:   LA STRINGA '4CD0' IN ASCII
                                *                                           OCCUPA LE LOC.6002-5
                                *
00004612: 48E7 E080             BINHEX   MOVEM.L  D0-D2/A0,-(A7)   SALVA I REGISTRI USATI IN BINHEX
00004616: 7203                           MOVEQ    #3,D1            CONT. LOOP := 4-1
00004618: 342F 0014                      MOVE.W   16+4(A7),D2      PRENDI VALORE
0000461C: 206F 0016                      MOVEA.L  16+6(A7),A0      PRENDI IND. DELLA STRINGA
00004620: D1FC 0000
00004624: 0004                           ADDA.L   #4,A0            PUNT. OLTRE LA FINE DELLA STRINGA
                                ;
00004626: 1002                  LOOP     MOVE.B   D2,D0
00004628: 0200 000F                      ANDI.B   #$0F,D0          PRENDI NIBBLE DI ORDINE BASSO
0000462C: 4EB9 0000
00004630: 464A                           JSR      HEXDIGIT         CONVERTI IN UN CARATTERE ASCII
00004632: 1100                           MOVE.B   D0,-(A0)         SALVA CIFRA ASCII
00004634: E84A                           LSR.W    #4,D2            SHIFTA D2 PER AVERE NIBBLE SUCC.
00004636: 51C9 FFEE                      DBRA     D1,LOOP          RIPETI PER TUTTE LE 4 CIFRE
                                ;
0000463A: 4CDF 0107                      MOVEM.L  (A7)+,D0-D2/A0   RIPRISTINA VALORI INIZIALI DEI REG.
0000463E: 2F57 0006                      MOVE.L   (A7),6(A7)       SPOSTA IN BASSO IND. DI RITORNO
00004642: DFFC 0000
00004646: 0006                           ADDA.L   #6,A7            AGGIUSTA PUNT. STACK A IND. RITORNO
00004648: 4E75                           RTS
                                ;
                                ;
0000464A: 0C00 000A             HEXDIGIT CMP.B    #$0A,D0          CIFRA DECIMALE O LETTERA ESADEC.?
0000464E: 6D02                           BLT.S    ADDZ             SE E' UNA CIFRA VAI A ADDZ
00004650: 5E00                           ADD.B    #'A'-'0'-$0A,D0  OFFSET PER LE LETTERE
00004652: 0600 0030             ADDZ     ADD.B    #'0',D0          CONVERTI IN ASCII
                                ;
00004656: 4E75                           RTS

                                         END      BINHEX
```

Spiegazione del programma 11-2

Questo programma mostra un altro metodo per il trasferimento dei parametri: invece dei registri è stato impiegato lo stack. Perciò, all'inizio della subroutine, lo stack si presenterà in questo modo:

Parametro Indirizzo	(32 bit)	
Parametro Cifra Esadecimale	(16 bit)	
Indirizzo di Ritorno	(32 bit)	Punt. Stack di Sistema(A7)

Il puntatore allo stack di sistema (A7) si comporta, di solito, come un qualunque altro registro. Tuttavia, poichè tutti i riferimenti alle word ed alle long word devono essere allineati con l'inizio di una

word, l'MC68000 prende delle particolari precauzioni per garantire questo allineamento. Perciò, ***tutti* i dati salvati o prelevati dallo stack di sistema sono allineati con l'inizio di una word, anche i dati di un byte.** In quest'ultimo caso, il dato viene memorizzato nel byte di ordine alto (più significativo) di una word, mentre il byte di ordine basso (meno significativo) resta immutato.

A differenza del primo esempio, BINHEX altera il contenuto di quei registri dati e indirizzi, che non sono utilizzati per trasferire il risultato della subroutine. In certi casi, una inattesa alterazione dei registri da parte di una subroutine può determinare delle conseguenze imprevedibili nel programma chiamante. **È buona abitudine stabilire quali registri saranno interessati durante l'esecuzione di una subroutine.** È quanto abbiamo fatto per la subroutine BINHEX, nella descrizione introduttiva.

Subroutines Nidificate

Un altro modo per prevenire i problemi creati da una modifica dei registri è di salvare tutti i registri utilizzati da una subroutine, ripristinandoli al momento dell'uscita. **L'istruzione MOVEM (Move Multiple) rappresenta un mezzo molto efficace per il salvataggio ed il ripristino dei registri. Ogni volta che devono essere salvati o ripristinati due o più registri indice (dati o indirizzi), l'impiego dell'istruzione MOVEM garantisce sempre un risparmio di memoria. In termini di tempo, risulta generalmente preferibile l'impiego di MOVEM, quando si tratta di salvare due o più registri indice e quando se ne devono ripristinare almeno tre.** L'ordine in cui avviene il trasferimento dei registri indice con l'istruzione MOVEM dipende dal tipo di indirizzamento: nel caso del postincremento, i registri sono salvati a partire dal registro dati 0 fino al registro dati 7, quindi dal registro indirizzi 0 fino al 7; con il predecremento, viene seguito l'ordine inverso, a partire dal registro indirizzi 7. Perciò, dopo l'esecuzione della prima istruzione MOVEM in BINHEX, lo stack di sistema si presenterà in questo modo:

Parametro Indirizzo	(32 bit)
Parametro Cifra Esadecimale	(16 bit)
Indirizzo di Ritorno	(32 bit)
A0	(32 bit)
D2	(32 bit)
D1	(32 bit)
D0 (32 bit)	Puntatore Stack di Sistema (A7)

I parametri non sono trasferiti mediante i registri, ma vengono prelevati dallo stack di sistema. Bisogna fare attenzione nel prelevare i parametri, perchè altri elementi sono presenti sullo stack: l'istruzione MOVEM vi mette 16 byte, mentre JSR memorizza sullo stack i 4 byte dell'indirizzo di ritorno (0000460E nel nostro caso). MOVE.W 16+4 (A7), A0 carica in A0 i 32 bit dell'indirizzo della stringa. L'ordine di queste due istruzioni non ha importanza, dato che non viene interessato il registro che agisce come puntatore allo stack di sistema.

Entrambe queste istruzioni MOVE rappresentano due esempi di indirizzamento indiretto a registro indirizzi con spostamento. È un indirizzamento simile a quello relativo al contatore di programma

con spostamento impiegato dall'istruzione di salto, ma con due differenze sostanziali: innanzitutto, viene utilizzato un registro indirizzi, invece del contatore di programma; in secondo luogo, è consentito soltanto un valore di spostamento a 16 bit, anche se con estensione del segno. Nel Programma 11-2, il puntatore allo stack di sistema contiene, all'inizio di MAIN, $7FFC. Perciò, l'indirizzo cui si riferisce la prima istruzione MOVE è:

```
(A7) + 16 + 4
= $7FE2 + 16 + 4
= $7FF6 (l'indirizzo della cifra)
```

Prima di restituire il controllo al programma MAIN è necessario ripristinare lo stack di sistema. Per prima cosa, preleviamo i registri mediante l'istruzione MOVEM (A7)+,D0-D2/A0. Quindi, ritorniamo a MAIN, servendoci di un'istruzione RTS, in quanto l'indirizzo di ritorno si trova alla sommità dello stack. Tuttavia, in questo modo sullo stack resterebbero ancora i parametri ed il programma principale dovrebbe provvedere al relativo aggiustamento, un compito da assolvere dopo ogni chiamata della subroutine BINHEX. Invece, di questo aggiustamento dello stack di sistema si occupa BINHEX, mediante la sequenza di istruzioni:

```
MOVE.L (A7),6(A7)
ADDA 06,A7
```

Grazie alla possibilità di trasferimenti da memoria a memoria, l'indirizzo di ritorno è memorizzato sullo stack nella posizione, che, in precedenza, era occupata dal parametro dell'indirizzo; quindi, viene modificato il puntatore allo stack, affinchè indichi questa nuova posizione. Si otterrebbe lo stesso risultato, e con maggior rapidità, sostituendo l'istruzione ADDA con un'istruzione LEA 6(A7),A7. Questo è il quadro dello stack prima e dopo le istruzioni MOVE e ADDA:

```
Prima:
(A7) →    7FF2     – 0000460E (indirizzo di ritorno)
          7FF6     – 4CD0 (parametro valore)
          7FF8     – 00006002 (parametro indirizzo)

Dopo:
          7FF2     – 0000460E
          7FF6     – 4CD0
(A7) →    7FF8     – 0000460E (indirizzo di ritorno)
```

Se anche i risultati fossero trasferiti tramite lo stack sarebbe necessario un diverso tipo di aggiustamento.

Questa subroutine è, allo stesso tempo, rientrante e indipendente dalla posizione, dal momento che non utilizza degli indirizzi fissi e si serve di istruzioni di salto relative.

Le istruzioni BSR e JSR permettono l'impiego di subroutine nidificate, in quanto successive chiamate metteranno il loro indirizzo di ritorno in posizioni sempre più basse dello stack. Nessun indirizzo va perso e l'istruzione RTS restituisce sempre il controllo all'istruzione immediatamente successiva alla più recente BSR o JSR.

11-3. ADDIZIONE A 64 BIT

Scopo: Sommare due valori a 64 bit (4 word) e tornare al programma principale con il risultato nei registri dati D0 e D1. D0 conterrà la word più significativa del risultato.

Problema Campione:

Input:	Value 1	– $0420147AEB529CB8
	Value 2	– $3020EB8520473118
Risultato:	D0	– 34410000
	D1	– 0B99CDD0

Programma 11-3a

```
00006000:                 DATA     EQU      $6000
00004600:                 PROGRAM  EQU      $4600
                          ;
                                   ORG      DATA

                                   ORG      PROGRAM

00004600: 4EB9 0000
00004604: 4618            MAIN     JSR      ADD64              ADDIZIONE A 64 BIT
00004606:                          DC.L     $1,$12345678       PRIMO PARAMETRO
0000460E:                          DC.L     $1,$12345          SECONDO PARAMETRO
00004616: 4E75                     RTS
                          *
                          * SUBROUTINE ADD64
                          *
                          * SCOPO:                SOMMA DUE VALORI A 64 BIT
                          *
                          * CONDIZIONI INIZIALI: I DUE PARAMETRI SI TROVANO SUBITO
                          *                       DOPO LA CHIAMATA DELLA SUBROUTINE
                          *
                          * CONDIZIONI FINALI:    LA SOMMA DEI DUE PARAMETRI A 64 BIT
                          *                       RITORNA IN D0.L E D1.L. IL COD.COND. EXTEND
                          *                       E' = 1 IN CASO DI OVERFLOW, ALTRIMENTI = 0
                          *
                          * USO DEI REGISTRI:     NESSUN REGSITRO TRANNE D0 E D1
                          *
                          * CASO CAMPIONE:        CONDIZIONI INIZIALI: 1^PARAM. = $112345678
                          *                                            2^PARAM. = $100012345
                          *                       CONDIZIONI FINALI:   D0.L = $00000002
                          *                                            D1.L = $123579BD
                          *                                            CC.X = 0
                          *
00004618: 48E7 3080       ADD64    MOVEM.L D2-D3/A0,-(A7)    SALVA D2, D3 E A0
0000461C: 206F 000C                MOVEA.L 12(A7),A0         A0 - IND. DEL PRIMO PARAMETRO
00004620: 4CD8 000F                MOVEM.L (A0)+,D0-D3       D0-D1 = PRIMO VAL., D2-D3 = SECONDO
                          ;
00004624: D283                     ADD.L   D3,D1             SOMMA WORD MENO SIGNIFICATIVA
00004626: D182                     ADDX.L  D2,D0             SOMMA I 16 BIT PIU' SIGN. CON EXT.
                          ;
00004628: 4CDF 010C                MOVEM.L (A7)+,D2-D3/A0    RIPRISTINA D2,D3 E A0
0000462C: 40E7                     MOVE.W  SR,-(A7)          SALVA FLAG DI EXTEND
0000462E: 06AF 0000
00004632: 0010 0002                ADDI.L  #16,2(A7)         CORREGGI INDIRIZZO DI RITORNO
00004636: 4E77                     RTR                       RITORNA E RIPRISTINA FLAG EXTEND

                                   END     ADD64
```

Spiegazione del programma 11-3a

Nel Programma 11-3a, i parametri per la subroutine ADD64 si trovano immediatamente dopo l'istruzione di chiamata. Al momento dell'ingresso in ADD64, l'indirizzo di questo blocco di parametri è alla sommità dello stack di sistema, dato che si tratta dello stesso indirizzo di ritorno per l'istruzione JSR. L'istruzione MOVEA.L

carica questo indirizzo nel registro A0. Lo spostamento di 12 si spiega con il fatto che sullo stack di sistema sono stati messi tre registri a 32 bit.

Il processo di addizione vero e proprio è molto semplice ed è stato descritto nel Capitolo 8. Prima di ritornare al programma principale, MAIN, l'indirizzo di ritorno deve essere modificato in modo da puntare all'istruzione che segue JSR. Per aggirare i due parametri da 8 byte, è necessario un aggiustamento di 16 byte . A questo provvede l'istruzione ADDI, che agisce sull'indirizzo di ritorno senza doverlo prima spostare in un registro. Ecco come si presenta lo stack di sistema prima e dopo l'istruzione ADDI:

```
Prima:
 (A7) (7FF6) = Registro di Stato (16 bit)
      (7FF8) = 4604

Dopo:
 (A7) (7FF6) = Registro di Stato
      (7FF8) = 4614
```

Dopo l'addizione che modifica opportunamente il puntatore, il registro di stato viene messo sullo stack allo scopo di salvare i codici di condizione. Questo consente al programma principale di controllare la presenza di eventuali riporti od overflow, verificatisi durante l'addizione a 64 bit. Un tale controllo è, normalmente, eseguito da un'istruzione di "salto condizionato", immediatamente successiva a JSR o alla lista dei parametri. In questo caso, i codici di condizione devono essere salvati, poichè il loro stato potrebbe essere alterato da ADDI. Proprio per un'eventualità come questa, l'MC68000 dispone di una speciale istruzione di ritorno: RTR (ritorna e ripristina i codici di condizione), che preleva dallo stack sia i codici di condizione che l'indirizzo di ritorno. La parte supervisore del registro di stato non è interessata da questa istruzione, che si rivela estremamente utile quando ci serviamo dei codici di condizione per indicare eventuali errori verificatisi in una subroutine.

L'istruzione RTR

Si presuppone, in linea generale, che la chiamata di una subroutine modifichi i codici di condizione, a meno che non sia indicato altrimenti. Se il programma principale avesse avuto necessità dei vecchi codici di condizione (per un successivo controllo), li avrebbe potuti salvare sullo stack di sistema, usando MOVE SR,-(A7) prima di chiamare la subroutine, e ripristinare successivamente con MOVE (A7)+,CCR.

Questo programma non è abbastanza generico, in quanto i valori dei parametri si trovano subito dopo la chiamata della subroutine, per cui, se il programma si trovasse nella memoria di sola lettura, una loro modifica risulterebbe praticamente impossibile. Per superare questo problema, forniremo alla subroutine gli indirizzi dei parametri, invece del loro valore.

Il Programma 11-3b indica le modifiche da apportare al programma precedente.

Programma 11-3b

```
00006000:                    DATA     EQU     $6000
00004600:                    PROGRAM  EQU     $4600
                             ;
                                      ORG     DATA

00006000:                    VALUE1   DS.L    2                   PRIMO VALORE A 64 BIT
00006008:                    VALUE2   DS.L    2                   SECONDO VALORE A 64 BIT
                             ;
                                      ORG     PROGRAM

00004600: 4EB9 0000
00004604: 4610               MAIN     JSR     ADD64               ADDIZIONE A 64 BIT
00004606:                             DC.L    VALUE1              IND. DEL PRIMO PARAMETRO
0000460A:                             DC.L    VALUE2              IND. DEL SECONDO PARAMETRO
0000460E: 4E75                        RTS
                             *
                             * SUBROUTINE ADD64
                             *
                             * SCOPO:                 SOMMA DUE VALORI A 64 BIT
                             *
                             * CONDIZIONI INIZIALI:   I DUE PARAMETRI SI TROVANO SUBITO
                             *                        DOPO LA CHIAMATA DELLA SUBROUTINE
                             *
                             * CONDIZIONI FINALI:     LA SOMMA DEI DUE PARAMETRI A 64 BIT
                             *                        RITORNA IN D0.L E D1.L. IL COD.COND. EXTEND
                             *                        E' = 1 IN CASO DI OVERFLOW, ALTRIMENTI = 0
                             *
                             * USO DEI REGISTRI:      NESSUN REGISTRO E' MODIFICATO TRANNE D0 E D1
                             *
                             * CASO CAMPIONE:         CONDIZIONI INIZIALI: 1^PARAM. = $00006000
                             *                                             2^PARAM. = $00006004
                             *                                         ($6000) = $0420147AEB529CB8
                             *                                         ($6004) = $3020EB8520473118
                             *
                             *                        CONDIZIONI FINALI:   D0.L = $34410000
                             *                                             D1.L = $0B99CDD0
                             *                                             CC.X = 0
                             *
00004610: 48E7 30C0          ADD64    MOVEM.L D2-D3/A0-A1,-(A7)   SALVA D2, D3, A0 E A1
00004614: 206F 0010                   MOVEA.L 16(A7),A0           A0 - IND. DEL BLOCCO PARAMETRI
                             ;
00004618: 2258                        MOVEA.L (A0)+,A1            A1-IND .DEL PRIMO PARAMETRO
0000461A: 2029 0000                   MOVE.L  0(A1),D0            WORD PIU' SIGN. DEL PRIMO VALORE
0000461E: 2229 0004                   MOVE.L  4(A1),D1            ...E MENO SIGNIFICATIVA
                             ;
00004622: 2258                        MOVEA.L (A0)+,A1            A1-IND. DEL SECONDO PARAMETRO
00004624: 2429 0000                   MOVE.L  0(A1),D2            WORD PIU' SIGN. DEL SECONDO VALORE
00004628: 2629 0004                   MOVE.L  4(A1),D3            ...E MENO SIGNIFICATIVA
                             ;
0000462C: 2F48 0010                   MOVEA.L A0,16(A7)           AGGIORNA INDIRIZZO DI RITORNO
00004630: D283                        ADD.L   D3,D1               SOMMA WORD MENO SIGNIFICATIVA
00004632: D182                        ADDX.L  D2,D0               SOMMA WORD PIU' SIGNIFICATIVA
                             ;
00004634: 4CDF 030C                   MOVEM.L (A7)+,D2-D3/A0-A1 RIPRISTINA REGISTRI UTILIZZATI
                             ;
00004638: 4E75                        RTS

                                      END     ADD64
```

Spiegazione del programma 11-3b

Le istruzioni in 11-3b sono, sostanzialmente, le stesse del Programma 11-3a. Tuttavia, una volta determinato l'indirizzo del blocco parametri (MOVE.L 16(A7),A0), è necessaria un'altra istruzione per ottenere i valori corrispondenti:

```
MOVEA.L   (A0)+,A1   Prendi l'indirizzo del parametro
MOVE.L    0(A1),D0   Prendi il valore
MOVE.L    4(A1),D1   ...del parametro
```

L'uso dell'indirizzamento con postincremento per prelevare gli indirizzi dei parametri serve anche ad aggiornare l'indirizzo di ritorno. Dopo le due istruzioni MOVE.L (A0)+,A1, il registro A0 contiene l'esatto indirizzo di ritorno, che verrà sostituito a quello presente sullo stack di sistema: (MOVEA.L A0,16(A7)). Questo consente di eliminare l'istruzione ADDI e, quindi, non sarà più necessario mettere i codici di condizione sullo stack di sistema.

11-4. FATTORIALE DI UN NUMERO

Scopo: Determinare il fattoriale del numero contenuto nella variabile NUMB, alla locazione di memoria 6000. Salvare il risultato nella variabile FNMB, alla locazione 6002. Si presuppone che il numero sia minore di nove e maggiore di zero.

Problemi Campione:

a.	Input:	NUMB-(6000)	= 0002
	Result:	FNUMB-(6002)	= 0002
b.	Input:	NUMB-(6000)	= 0005
	Result:	FNUMB-(6002)	= 0078(120_{10})

Diagramma di Flusso 11-4

Programma 11-4a

```
00006000:                    DATA     EQU      $6000
00004600:                    PROGRAM  EQU      $4600
                             ;
                                      ORG      DATA

00006000:                    NUMB     DS.W     1              NUMERO
00006002:                    F_NUMB   DS.W     1              FATTORIALE DEL NUMERO
                             ;
                                      ORG      PROGRAM

00004600: 3038 6000          MAIN     MOVE.W   NUMB,D0        PRENDI IL NUMERO
00004604: 6106                        BSR.S    FACTOR         TROVA FATTORIALE
00004606: 31C0 6002                   MOVE.W   D0,F_NUMB      SALVA FATTORIALE
```

```
0000460A: 4E75          ;       RTS
                        *
                        * SUBROUTINE FACTOR
                        *
                        * SCOPO:                DETERMINARE FATTORIALE DI UN NUMERO
                        *
                        * CONDIZIONI INIZIALI: D0.W = NUM. DI CUI VA TROVATO IL
                        *                       FATTORIALE. D0.W > 0 E < 9
                        *
                        * CONDIZIONI FINALI:    D0.W = FATTORIALE DEL NUMERO
                        *
                        * USO DEI REGISTRI:     NESSUN REGISTRO TRANNE D0
                        *
                        * CASO CAMPIONE:        CONDIZIONI INIZIALI: D0.W = 5
                        *                       CONDIZIONI FINALI:   D0.W = 120
                        *
0000460C: 3F00          FACTOR  MOVE.W  D0,-(A7)        METTI IL NUMERO SULLO STACK
0000460E: 5340                  SUBQ.W  #1,D0           DECREMENTA IL NUMERO
00004610: 6604                  BNE.S   F_CONT          NON ANCORA FINITO
                        ;
00004612: 301F                  MOVE.W  (A7)+,D0        FATTORIALE := 1
00004614: 6004                  BRA.S   RETURN
                        ;
00004616: 61F4          F_CONT  BSR     FACTOR
00004618: C0DF                  MULU    (A7)+,D0        FATTORIALE := N * (N-1)
                        ;
0000461A: 4E75          RETURN  RTS
                                END     FACTOR
```

Questa è una subroutine rientrante, dal momento che non usa un'area fissa per la memorizzazione dei dati, ma ad essi viene riservato uno spazio sullo stack. È, inoltre, ricorsiva, perchè richiama se stessa con l'istruzione BSR FACTOR.

Quelle ricorsive rappresentano un caso particolare di subroutine nidificate. Come per ogni altra chiamata di subroutine fatta mediante un'istruzione BSR o JSR, l'indirizzo di ritorno viene messo alla sommità allo stack; in questo caso, il processore non si preoccupa minimante del fatto che vi siano due indirizzi di ritorno identici.

La subroutine FACTOR è un esempio molto semplice di routine ricorsiva poichè, come potete notare, FACTOR richiama se stessa.

Ricorsività Indiretta

Una subroutine è ricorsiva anche nel caso in cui richiama una routine che, a sua volta, invoca la subroutine chiamante. Una ricorsività di questo tipo viene detta "indiretta", in contrapposizione a quella definita "diretta" in cui una subroutine richiama direttamente se stessa. Ad esempio, FACTOR sarebbe ricorsiva indiretta se:

```
F CONT:  BSR FACTOR
         MULU (A7)+,D0
```

fossero sostituite con:

```
F CONT: BSR MULTIPLY
```

dove MULTIPLY è una subroutine di questo tipo:

```
MULTIPLY: BSR FACTOR
          MULU (A7)+,D0
          RTS
```

Come qualunque subroutine che usi lo stack per funzioni di memorizzazione temporanea, FACTOR *deve* assicurarsi che sullo stack non siano rimasti dei dati prima dell'istruzione di ritorno: a questo provvedono le istruzioni MOVE.W (A7)+,D0 e MULU (A7)+,D0.

In molti casi, capita di non essere certi delle esatte condizioni dello stack prima del ritorno. Ciò è vero soprattutto per coloro che hanno

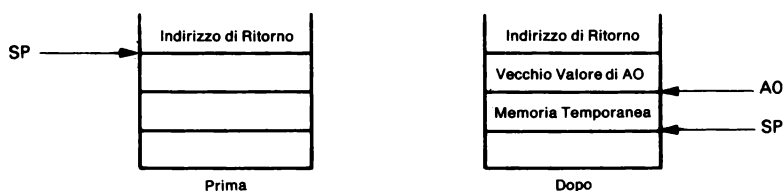
delle buone abitudini di programmazione e usano una sola istruzione di uscita o di ritorno (come nella subroutine FACTOR). Ma un aspetto ancor più importante è che, spesso, l'esecuzione di una subroutine non salva dei dati sullo stack in modo ordinato. Per questi motivi l'MC68000 dispone delle istruzioni LINK e UNLK.

Le istruzioni LINK e UNLK

La subroutine FACTOR è stata riscritta nel Programma 11-4b utilizzando LINK e UNLK. Con l'aiuto dell'istruzione LINK possiamo riservare, in modo dinamico, uno spazio di 32.678 byte sullo stack e, allo stesso tempo, definire un puntatore che indichi l'inizio dell'area riservata. Inoltre, l'istruzione LINK salva il valore attuale del puntatore.

Programma 11-4b

```
00006000:                    DATA     EQU     $6000
00004600:                    PROGRAM  EQU     $4600
                             ;
                                      ORG     DATA

00006000:                    NUMB     DS.W    1               NUMERO
00006002:                    F_NUMB   DS.W    1               FATTORIALE DEL NUMERO
                             ;
                                      ORG     PROGRAM

00004600: 3038 6000          MAIN     MOVE.W  NUMB,D0         PRENDI IL NUMERO
00004604: 6106                        BSR.S   FACTOR          TROVA FATTORIALE
00004606: 31C0 6002                   MOVE.W  D0,F_NUMB       SALVA FATTORIALE
                             ;
0000460A: 4E75                        RTS
                             *
                             * SUBROUTINE FACTOR
                             *
                             * SCOPO:                 DETERMINARE FATTORIALE DI UN NUMERO
                             *
                             * CONDIZIONI INIZIALI:   D0.W = NUM. DI CUI VA TROVATO IL
                             *                        FATTORIALE. D0.W > 0 E < 9
                             *
                             * CONDIZIONI FINALI:     D0.W = FATTORIALE DEL NUMERO
                             *
                             * USO DEI REGISTRI:      NESSUN REGISTRO TRANNE D0
                             *
                             * CASO CAMPIONE:         CONDIZIONI INIZIALI: D0.W = 5
                             *                        CONDIZIONI FINALI:   D0.W = 120
                             *
0000460C: 4E50 FFFE          FACTOR   LINK    A0,#-2          RISERVA SPAZIO TEMP. SULLO STACK
00004610: 3140 FFFE                   MOVE.W  D0,-2(A0)       SALVA NUMERO
00004614: 5340                        SUBQ.W  #1,D0           DECREMENTA IL NUMERO
00004616: 6604                        BNE.S   F_CONT          NON ANCORA FINITO
                             ;
00004618: 7001                        MOVEQ   #1,D0           FATTORIALE := 1
0000461A: 6006                        BRA.S   RETURN          RITORNA ALLA ROUTINE CHIAMANTE
                             ;
0000461C: 61EE               F_CONT   BSR     FACTOR          CONTINUA PROCESSO DI FATT.
0000461E: C0E8 FFFE                   MULU    -2(A0),D0       FATTORIALE := N * (N-1)
                             ;
00004622: 4E58               RETURN   UNLK    A0              LIBERA SPAZIO TEMPORANEO RISERVATO
                             ;
00004624: 4E75                        RTS

                                      END     FACTOR
```

Nel Programma 11-4b, l'istruzione LINK A0,#-2 ha l'effetto seguente:

L'istruzione UNLK ha effetti diametralmente opposti a quelli di un'istruzione LINK e serve a ripristinare lo stack e i registri indirizzi.

Quando utilizzate queste istruzioni ricordate che lo spostamento necessario per riservare uno spazio destinato alla memorizzazione

dei dati è uno spostamento negativo, in quanto lo stack si espande verso la parte bassa della memoria. Anche gli offset rispetto al registro puntatore dovranno essere negativi, dal momento che il registro indirizzi contiene l'indirizzo più alto dell'area riservata alla memorizzazione temporanea dei dati.

PROBLEMI

Per ogni problema proposto scrivere il programma chiamante e almeno una subroutine, documentata in modo appropriato.

11-1. DA ESADECIMALE ASCII A BINARIO

Scopo: Rappresentare in forma binaria (con 4 bit) la cifra esadecimale, il cui codice ASCII è indicato dagli otto bit meno significativi del registro dati D0. Mettere il risultato in D0.

Problemi Campione:

```
a. Input:      D0 = 43     'C'
   Risultato:  D0 = 0C
b. Input:      D0 = 36     '6'
   Risultato:  D0 = 06
```

11-2. DA STRINGA ESADECIMALE ASCII A WORD BINARIA

Scopo: Convertire i quattro caratteri ASCII della variabile STRING, che inizia alla locazione di memoria 6002, in un valore binario a 16 bit. Salvare il valore nella variabile VALUE, alla locazione 6000. Scrivere una subroutine che prende l'indirizzo della stringa dallo stack e ritorna il valore sempre sullo stack.

Problema Campione:

```
Input:       STRING    - (6002) = 42 'B'
                         (6003) = 32 '1'
                         (6004) = 46 'F'
                         (6005) = 30 '0'

Risultato:   VALUE     - (6000) = B1F0
```

11-3. VERIFICARE LA PRESENZA DI UN CARATTERE ALFABETICO

Scopo: Se il carattere ASCII nella variabile CHAR, alla locazione di memoria 6000, è alfabetico (maiuscolo o minuscolo), mettere FF_{16} nella variabile FLAG, alla locazione 6001; altrimenti, mettere FLAG a 0. Scrivere una subroutine che trovi il suo parametro in un registro e fornisca il risultato mediante i flag dei codici di condizione.

Problemi Campione:

```
a. Input:      CHAR    - (6000) = 47 'G'
   Risultato:  FLAG    - (6001) = FF

b. Input:      CHAR    - (6000) = 36 '6'
   Risultato:  FLAG    - (6001) = 00

c. Input:      CHAR    - (6000) = 6A 'j'
   Risultato:  FLAG    - (6001) = FF
```

11-4. RICERCA DEL PRIMO CARATTERE NON ALFABETICO

Scopo: La variabile STRING, alla locazione di memoria 6000, contiene l'indirizzo di una stringa ASCII. Mettere l'indirizzo del primo carattere non alfabetico di questa stringa nella variabile ADDRESS, alla locazione 6002. Scrivere una subroutine che prenda l'indirizzo da un registro e metta il risultato nello stesso registro.

Problemi Campione:

```
a. Input:      STRING    - (6000) = 6100
                           (6100) = 43 'C'
                           (6101) = 61 'à
                           (6102) = 74 't'
                           (6103) = 0D  CR

   Risultato:  ADDRESS   - (6002) = 6103

b. Input:      STRING    - (6000) = 6100
                           (6100) = 32 '2'
                           (6101) = 50 'P'
                           (6102) = 49 'I'
                           (6103) = 0D  CR

   Risultato:  ADDRESS   - (6002) = 6100
```

11-5. CONTROLLO DELLA PARITÀ DI TIPO PARI

Scopo: La variabile LENGTH, alla locazione di memoria 6001, contiene la lunghezza in byte della variabile stringa STRING, che ha inizio alla locazione 6002. Se ciascun byte della stringa ha una parità pari, mettere a 0 la variabile FLAG, alla locazione 6000; se uno o più byte hanno una parità dispari, mettere FF_{16} in FLAG. Scrivere una subroutine che ottiene lunghezza e locazione dallo stack e ritorna con il risultato sempre allo stack.

Problemi Campione:

a. Input:	LENGTH	– (6001)	= 3
	STRING	– (6002)	= 47
		(6003)	= AF
		(6004)	= 18
Risultato:	FLAG	– (6000)	= 00
b. Input:	LENGTH	– (6001)	= 3
	STRING	– (6002)	= 47
		(6003)	= AF
		(6004)	= 19
Risultato:	FLAG	– (6000)	= FF, poichè 19 = 00011001 ha parità dispari

11-6. CONFRONTARE DUE STRINGHE

Scopo: Scrivere una subroutine (e un programma principale che la controlli) per il confronto di due stringhe ASCII. Il primo byte di ogni stringa ne indica la lunghezza. Fornire il risultato mediante i codici di condizione; cioè, il flag S verrà posto a 1, se la prima stringa è minore (precedente) in ordine alfabetico rispetto alla seconda; il flag Z sarà messo a 1, se, invece, le stringhe sono uguali. Nessun flag viene posto a 1 se la seconda stringa precede la prima. Notare che ABCD è, in ordine alfabetico, maggiore di ABC.

CAPITOLO 12

INPUT/OUTPUT

Due sono i problemi fondamentali nella progettazione delle routine di input/output: uno è in che modo interfacciare le periferiche con il computer e trasferire dati e segnali di stato e di controllo; l'altro, come indirizzare i dispositivi di I/O in modo che la CPU possa selezionarne uno specifico per il trasferimento di dati. Chiaramente, il primo problema, oltre ad essere più complesso, è anche più interessante. **Perciò, nelle pagine seguenti parleremo dell'interfacciamento delle periferiche,** lasciando il problema dell'indirizzamento ad un testo che si occupi dell'hardware.

I/O E MEMORIA

Il trasferimento di dati da e verso un dispositivo di I/O è, teoricamente, **analogo al trasferimento di dati da e verso la memoria, che potremmo anche considerare come un altro dispositivo di I/O.** Tuttavia, **la memoria costituisce un caso un pò particolare, per i seguenti motivi:**

1. Funziona quasi alla stessa velocità del processore.
2. Usa lo stesso tipo di segnali della CPU. I soli circuiti necessari per l'interfacciamento della memoria alla CPU sono i driver, i ricevitori e, in qualche caso, i traduttori di livello.
3. Non richede formati speciali o segnali di controllo, ma soltanto un impulso Read/Write.
4. Conserva automaticamente i dati che gli sono stati inviati.
5. La lunghezza di parola è la stessa del computer.

Problemi legati al trasferimento di dati

La maggior parte dei dispositivi di I/O non ha queste caratteristiche di efficienza. In certi casi funzionano a velocità molto più lente rispetto al processore; ad esempio, una telescrivente riesce a trasferire solamente 30 caratteri al secondo, mentre un processore, anche se lento, ne può trasferire fino a 10.000. **La velocità varia, comunque, moltissimo da una periferica all'altra:** alcuni sensori forniscono un nuovo valore ogni minuto, mentre i terminali video ed i floppy disk arrivano a 250.000 bit al secondo. Inoltre, **alcuni dispositivi di I/O richiedono segnali continui** (motori o termometri), **altri correnti invece di tensioni** (le vecchie telescriventi) **oppure tensioni a livelli molto diversi da quelle dei segnali usati dal processore** (display a scarica di

gas). In certi casi sono necessari anche dei formati speciali, dei protocolli o dei segnali di controllo e la lunghezza della parola può essere molto diversa rispetto a quella del computer. **Queste differenze fanno sì che ogni periferica presenti un suo particolare problema di interfacciamento, rendendo la progettazione delle routine di I/O un compito particolarmente gravoso.**

CLASSIFICAZIONE DEI DISPOSITIVI DI I/O

Vi forniremo una descrizione delle caratteristiche di questi dispositivi e del modo in cui interfacciarli, suddividendoli, più o meno grossolanamente, in tre categorie in base alla velocità di trasferimento dei dati:

Diversità tra i vari dispositivi di I/O

1. **Dispositivi lenti, che cambiano il loro stato non più di una volta al secondo.** Il cambiamento richiede tempi nell'ordine dei millisecondi o più. A questa categoria appartengono i display luminosi, gli interruttori, i relè e molti sensori ed attuatori meccanici.
2. **Dispositivi a media velocità, che trasferiscono dati a velocità comprese fra 1 e 10.000 bit al secondo.** Ne fanno parte le tastiere, le stampanti, i lettori di schede, i lettori e perforatori di nastro, i registratori a cassette, le normali linee di comunicazione e molti dei sistemi analogici per l'acquisizione di dati.
3. **Dispositivi ad alta velocita, che trasferiscono dati a velocità superiori a 10.000 bit al secondo.** Comprendono i nastri magnetici, i dischi magnetici, le stampanti le linee di comunicazione ad alta velocità e i terminali video.

INTERFACCIAMENTO DEI DISPOSITIVI LENTI

L'interfacciamento dei dispositivi lenti è piuttosto semplice. Sono, infatti, sufficienti pochi segnali di controllo, a meno che non si tratti di dispositivi multiplexed, in cui una sola porta ne gestisce più di uno come viene indicato nelle figure seguenti. I dati provenienti da dispositivi lenti non devono essere messi in un latch, dal momento che rimangono stabili per molto tempo; naturalmente questa necessità esiste, invece, per i dati in uscita dal calcolatore. Gli unici problemi, in fase di input, si hanno con le transizioni che si verificano durante la fase di lettura dei dati da parte del computer. A questo possiamo ovviare facendo ricorso a monostabili, latch di tipo "cross-coupled" o routine di ritardo via software.

Demultiplexer con contatore

Una singola porta è in grado di gestire parecchi dispositivi lenti. La Figura 12-1 mostra un demultiplexer che automaticamente dirige il successivo dato in uscita verso un nuovo dispositivo, contando le operazioni di output. Nella Figura 12-2 si può osservare una porta di controllo che fornisce degli input di selezione ad un demultiplexer. L'uscita dei dati, in questo caso, può avvenire in un ordine

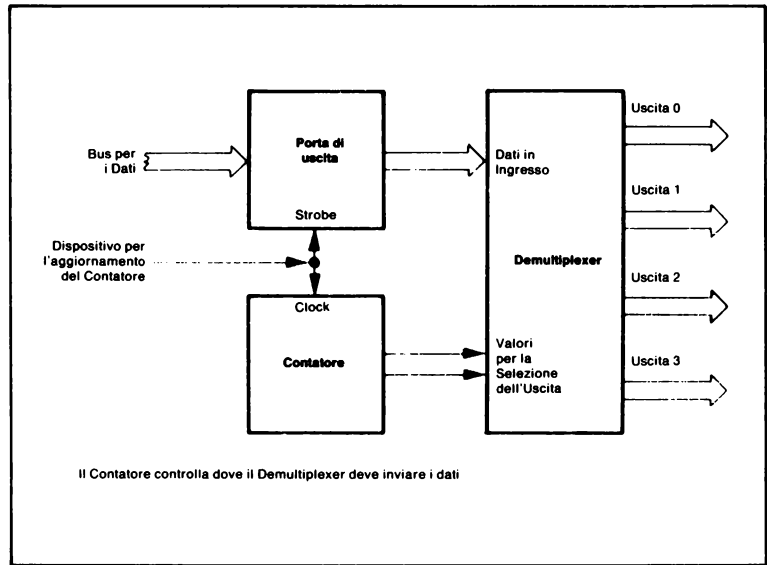



Figura 12-1. Demultiplexer di Uscita Controllato da un Contatore.

qualsiasi, ma ciò richiede un'ulteriore istruzione per cambiare lo stato della porta di controllo. I demultiplexer per dati in uscita sono comunemente usati per pilotare un certo numero di display tramite una stessa porta. Le figure 12-3 e 12-4 mostrano le stesse alternative nel caso di un multiplexer per dati in ingresso.

Demultiplexer con porta di controllo

Osservate le differenze fra la fase di input e quella di ouput con un dispositivo lento.

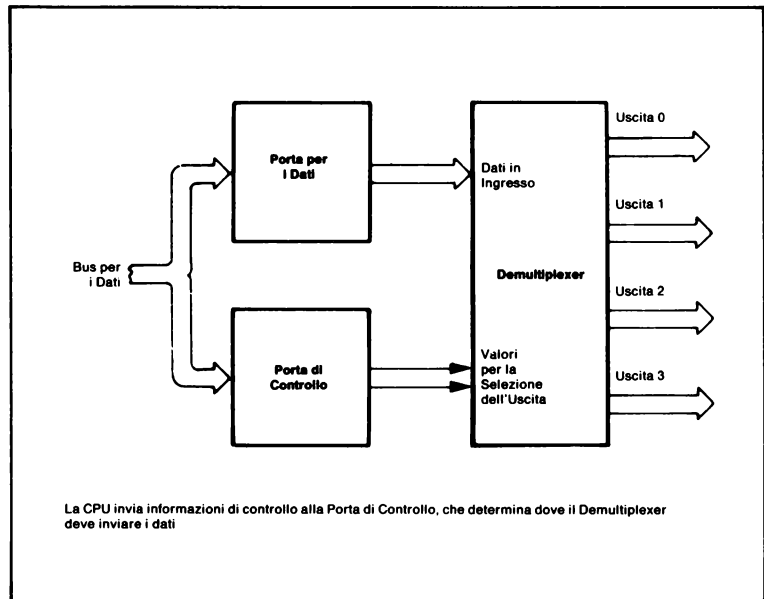



Figura 12-2. Un Demultiplexer di Uscita Controllato da una Porta.

Il Contatore controlla da quale Ingresso il Multiplexer deve prendere i dati da inviare alla Porta d'Ingresso

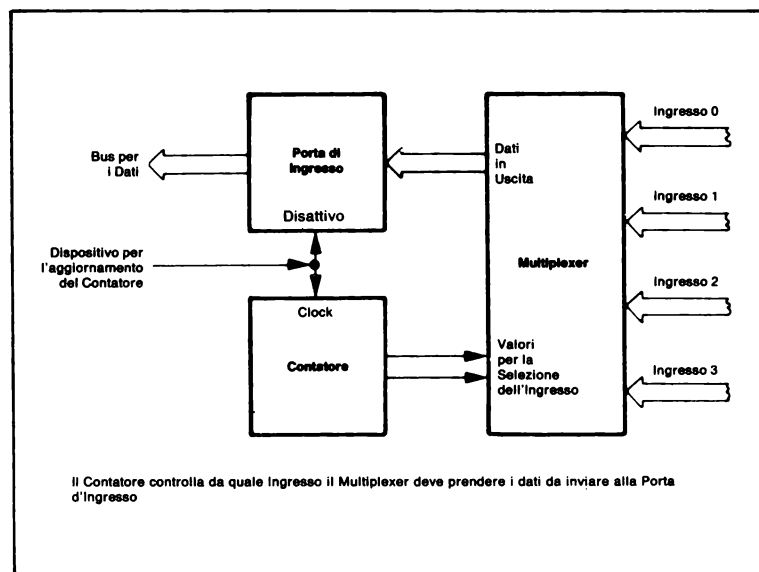



Figura 12-3. Un Multiplexer di Ingresso Controllato da un Contatore.

1. **Non sono necessari dei latch per i dati in ingresso,** in quanto il dispositivo di input li mantiene stabili per un periodo di tempo, che è estremamente lungo se raffrontato con gli standard di un computer. I dati in uscita richiedono, invece, dei latch, poichè il dispositivo di ouput non riconoscerà dei dati, che rimangono stabili soltanto per alcuni cicli di clock. Ricordate che la CPU utilizza continuamente il suo bus dati, per effettuare normali trasferimenti da e verso la memoria.

La CPU invia delle informazioni di controllo alla Porta di Controllo, che determina quale Ingresso deve essere connesso dal Multiplexer con la Porta per i Dati.

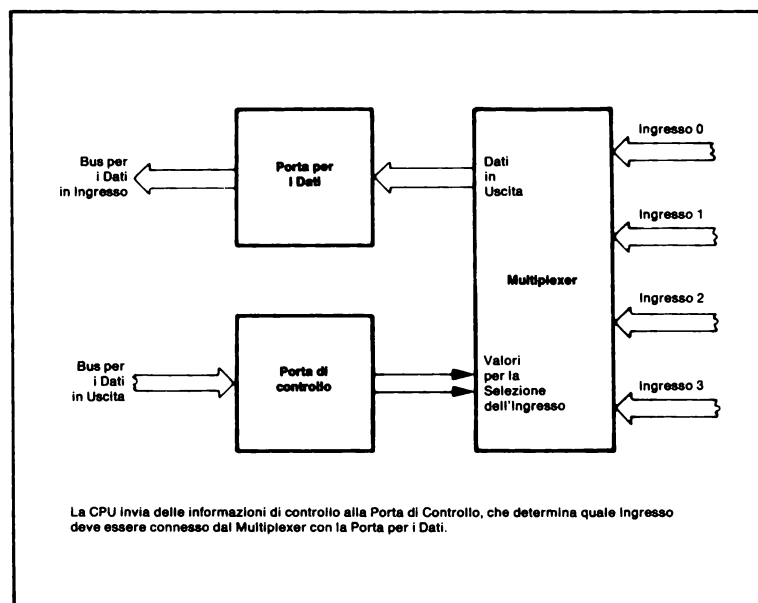



Figura 12-4. Un Multiplexer di Ingresso Controllato da una Porta.

2. **Le transizioni in ingresso danno luogo a dei problemi a causa della loro durata; mentre brevi transizioni in fase di output non danno alcun problema,** dato che i dispositivi di uscita (o eventuali osservatori) hanno tempi di reazione molto lunghi.
3. **Le maggiori limitazioni in fase di input sono costituite dai tempi di reazione e di risposta; quelle in fase di output dai tempi di risposta e dalla osservabilità.**

INTERFACCIAMENTO DEI DISPOSITIVI A MEDIA VELOCITÀ

I dispositivi a media velocità devono in qualche modo essere sincronizzati con il processore. La CPU non può semplicemente considerare questi dispositivi come se essi fossero in grado di mantenere stabili i loro dati per un tempo indefinito e riceverne di nuovi in qualsiasi momento. Deve, invece, avere la possibilità di sapere quando un dispositivo è pronto per inviare un nuovo dato oppure per riceverne uno in uscita. Inoltre, la CPU deve segnalare ad una periferica quando è disponibile un nuovo dato ed informarla dell'avvenuta ricezione di un input. **Tenete presente che una periferica, in certi casi, è costituita da un altro processore o, comunque, ne può contenere uno.**

Handshake

La procedura standard di sincronizzazione non controllata da un clock, è l'"handshake" (lett. stretta di mano). Il trasmettitore trasferisce il dato solo dopo averne segnalata al ricevitore la disponibilità; il ricevitore completa l'handshake comunicando l'avvenuta ricezione. Il ricevitore può assumere il controllo della situazione richiedendo per primo il dato o indicando la sua disponibilità a riceverlo; il trasmettitore, allora, invia il dato e completa l'handshake indicando che il dato è disponibile. In entrambi i casi, il trasmettitore sa che il trasferimento è avvenuto con successo ed il ricevitore sa quando è disponibile un nuovo dato. La procedura di handshake può funzionare ad una qualsiasi velocità, poichè sono il trasmettitore ed il ricevitore (e non un clock) a controllare la sequenza.

Polling

Le Figure 12-5 e 12-6 mostrano delle normali operazioni di input/output che fanno uso dell'handshake. La procedura con la quale la CPU controlla la disponibilità della periferica, prima del trasferimento di un dato, è chiamata "polling". Chiaramente, il polling può occupare gran parte del tempo del processore, soprattutto in presenza di un gran numero di dispositivi di I/O. **Esistono parecchi modi di fornire dei segnali di handshake. Eccone alcuni:**

- **Differenti linee di I/O dedicate.** Il processore può gestirle come porte di I/O addizionali oppure mediante delle linee speciali o degli interrupt. L'MC68000 non ha delle speciali linee di I/O, ma

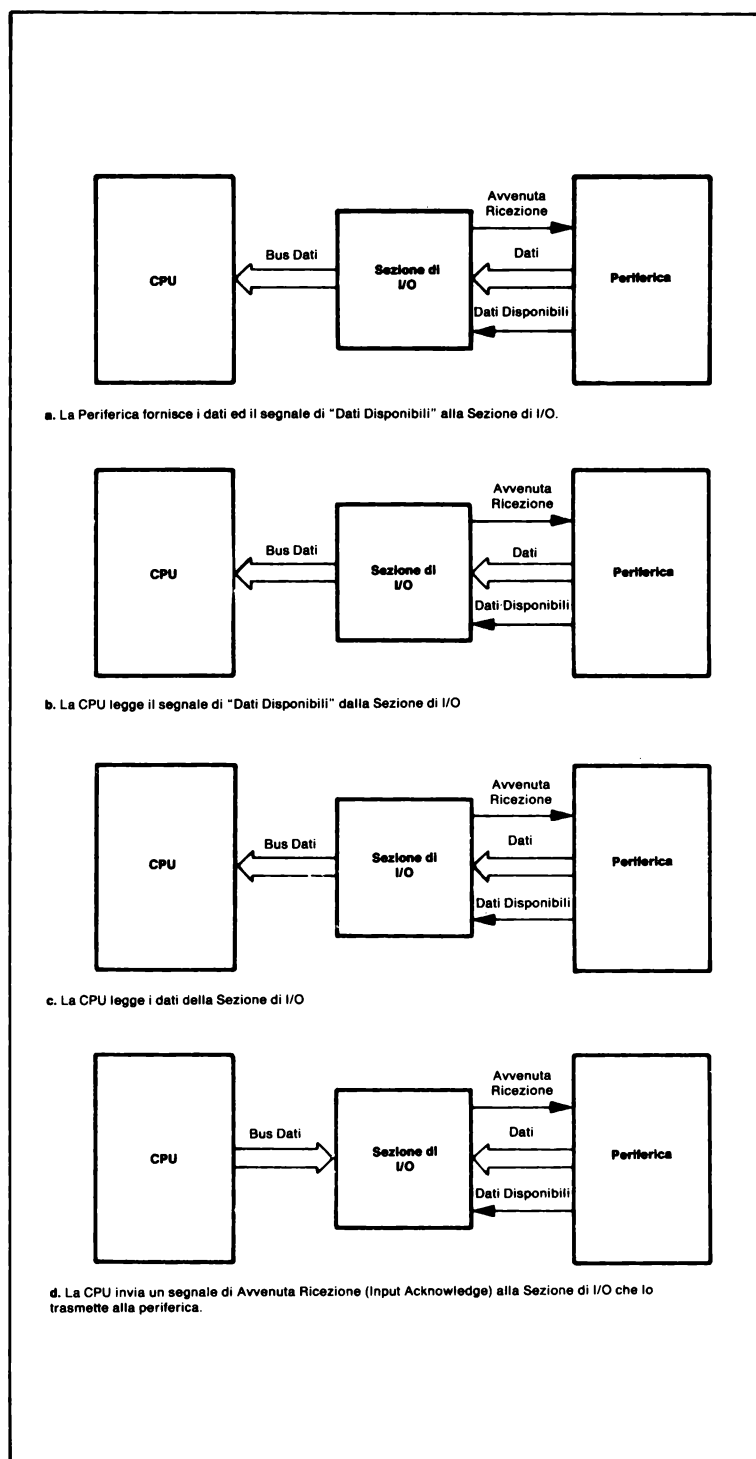



Figura 12-5. Un Handshake di Ingresso.

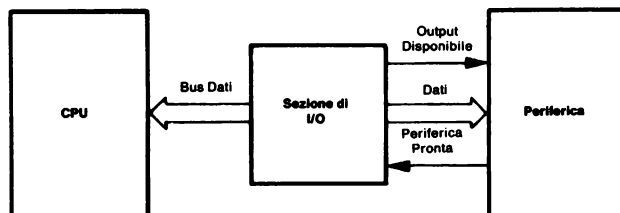
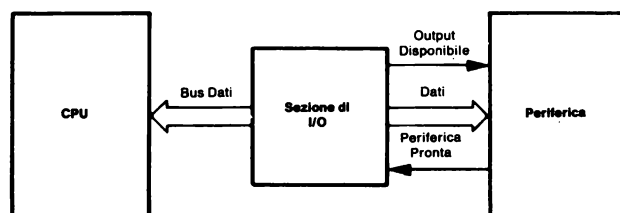
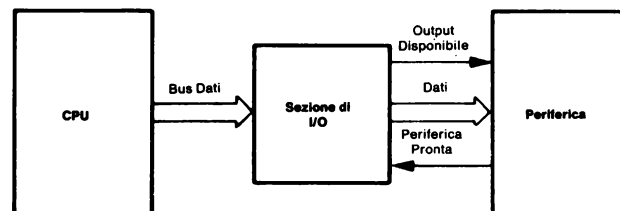
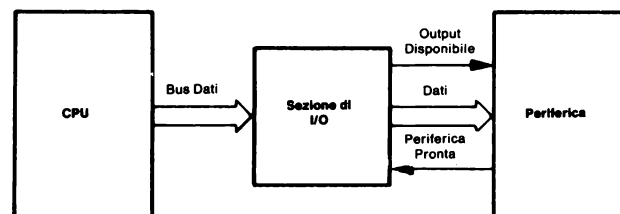



Figura 12-6. Un Handshake di Uscita.

a questo provvede il 6821 Peripheral Interface Adapter (o chip programmabile per l'interfacciamento parallelo).
- **Sequenze speciali sulle linee di I/O.** Può trattarsi di singoli bit di start e stop oppure di interi caratteri o gruppi di caratteri. Le sequenze devono essere facilmente distinguibili dal rumore di fondo o da stati di inattività.

Strobe

Chiamiamo "strobe" una linea separata di I/O che indica la disponibilità di un dato o il verificarsi di un trasferimento. Uno strobe può, ad esempio, mettere un dato in un latch o prelevarne uno da un buffer.

CLOCK

I/O Sincronizzata

Molte periferiche trasferiscono i dati ad intervalli regolari, cioè in modo sincronizzato. In questo caso, il solo problema è quello di iniziare il processo allineandosi al primo input o contrassegnando il primo output. Alcune volte la periferica fornisce un'uscita di clock da cui il processore può ottenere informazioni riguardo alla temporizzazione. **Nell'I/O sincronizzato i trasferimenti dei dati sono controllati da un clock e non da uno scambio di informazioni fra trasmettitore e ricevitore.**

Ridurre gli Errori di Trasmissione

Gli errori di trasmissione cosituiscono uno dei problemi più grossi con i dispositivi a media velocità. Questi sono solo alcuni dei metodi utilizzati per ridurne l'incidenza:

- **Il campionamento dei dati in entrata al centro dell'intervallo di trasmissione,** allo scopo di evitare gli effetti dei fronti; cioè, tenersi lontani dai fronti, dove i dati cambiano.
- **Un campionamento ripetuto di ogni singolo dato e l'impiego della logica maggioritaria.** Ad esempio, viene letto cinque volte lo stesso bit, prendendo il valore che risulta più frequentemente[1].
- **La generazione ed il controllo della parità;** si utilizza un altro bit[2], in modo da rendere pari o dispari il numero totale di bit 1 presenti in un dato corretto.
- **L'impiego di altri codici che consentano di individuare e correggere eventuali errori,** come checksum, LRC (controllo di ridondanza longitudinale) e CRC (controllo di ridondanza ciclica).[2]

INTERFACCIAMENTO DEI DISPOSITIVI AD ALTA VELOCITÀ

Accesso Diretto alla Memoria

I dispositivi ad alta velocità, che trasferiscono più di 10.000 bit al secondo, richiedono metodologie particolari. Normalmente viene utilizzato un controller, che trasferisce direttamente i dati dalla memoria al dispositivo di I/O e viceversa: è il cosiddetto processo di accesso diretto alla memoria (DMA). Un controller DMA toglie alla CPU il controllo dei bus, fornisce indirizzi e segnali di controllo alla memoria e trasferisce i dati. Un controller di questo tipo è abbastanza complesso essendo costituito, di solito, da 50-100 chip, sebbene dispositivi LSI, come il controller DMA MC68450[3] per l'MC68000, siano reperibili sul mercato fin dal 1981. La CPU deve inizialmente caricare i contatori dei dati e degli indirizzi sul controller, in modo che questo possa sapere da dove partire e quanti dati trasferire.

INTERVALLI DI TEMPO

Un problema che incontrerete molto spesso nella stesura di routine di I/O è quello relativo agli intervalli di tempo necessari fra un'operazione di I/O e quella successiva. Questi intervalli, di diversa durata, servono ad eliminare i rimbalzi degli interruttori meccanici (cioè, a smorzare le loro transizioni irregolari), a fornire impulsi di determinata lunghezza e frequenza ai display ed a sincronizzare le operazioni di I/O per quei dispositivi che trasferiscono i dati con una certa periodicità (ad es., una telescrivente che invia o riceve un bit ogni 9.1 ms.).

METODI PER OTTENERE DEI RITARDI

Questi ritardi possono essere realizzati in vari modi:

Fattori che influenzano la durata effettiva del ritardo

1. **Tramite hardware,** con monostabili o multivibratori monostabili. Questi dispositivi forniscono un singolo impulso di durata costante in risposta ad un impulso in entrata. Tuttavia, i monostabili creano dei problemi di affidabilità e dovrebbero, se possibile, essere evitati.
2. **Mediante una combinazione di hardware e software,** con un dispositivo flessibile come il Timer Programmabile 6840 nel caso dei microcomputer dotati dell'MC68000.[4] Il 6840 è in grado di fornire intervalli di tempo di varia durata, con una vasta gamma di condizioni iniziali e finali.

3. **Con delle apposite routine di ritardo,** che avranno il solo scopo di far passare del tempo. Per realizzare queste routine è necessario conoscere la frequenza di clock dell'elaboratore (è una caratteristica dipendente dal sistema) ed il numero di cicli di clock richiesti per l'esecuzione di ogni singola istruzione (Appendice B). Il problema con le routine di ritardo è che il processore non può svolgere altre funzioni, mentre lascia passare questo intervallo; tuttavia, esse non prevedono l'impiego di particolari componenti hardware e, spesso, utilizzano i momenti in cui il processore resterebbe, comunque, inattivo.

La scelta di uno di questi tre metodi dipende dal tipo di applicazione. Se ci serviamo di una routine di ritardo riduciamo i costi, ma rischiamo di sovraccaricare il processore. I timer programmabili comportano dei costi leggermente superiori, ma, in compenso, sono facili da interfacciare e ad essi possiamo delegare gran parte delle funzioni di temporizzazione.

ROUTINE DI RITARDO

Una semplice routine di ritardo funziona in questo modo:

1. **Carica un registro con un determinato valore.**
2. **Decrementa il registro.**
3. **Se il risultato non è zero, ripete la fase 2.**

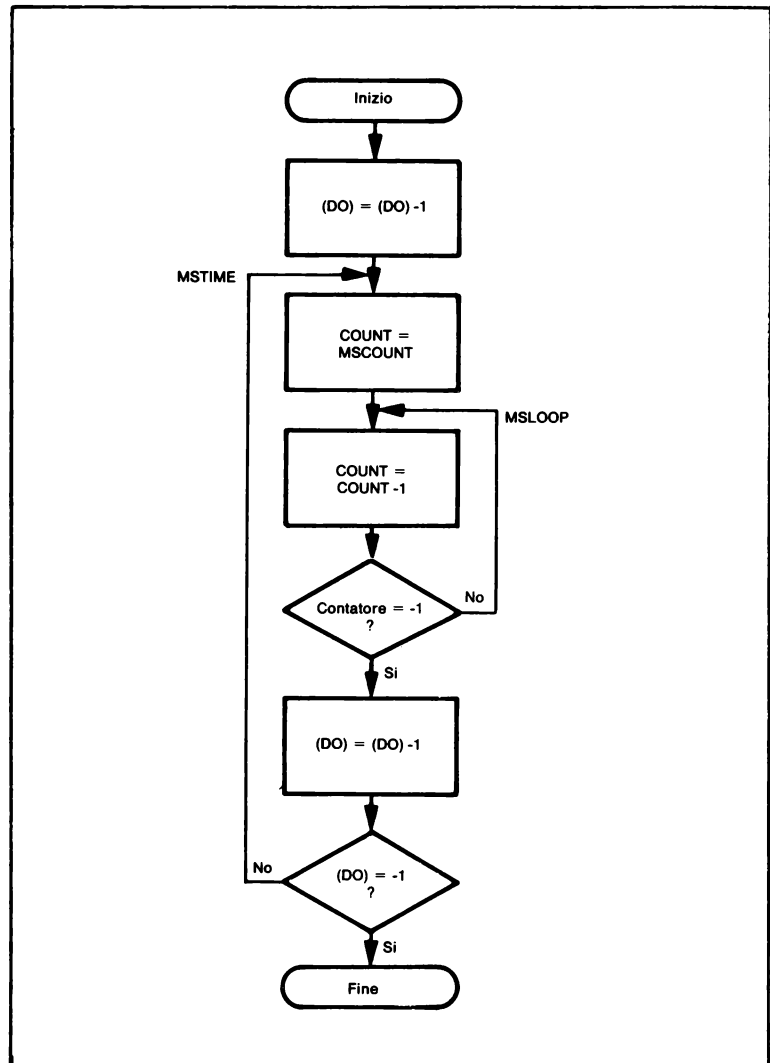
Questa routine fa soltanto trascorrere del tempo; la durata dipende dal tempo di esecuzione delle singole istruzioni. La lunghezza massima del ritardo dipende dalle dimensioni del registro; l'intera routine può, tuttavia, essere messa all'interno di un'altra simile che utilizza un altro registro, e così via.

Fate attenzione: **il tempo effettivamente trascorso dipende dalla frequenza del clock con cui sta funzionando il processore, dalla velocità di accesso alla memoria e dalle condizioni operative, come la temperatura, la tensione di alimentazione ed il carico dei circuiti, tutte cose che possono alterare l'esatta frequenza del clock di sistema.**

L'esempio seguente utilizza i registri D0 e D1 per generare dei ritardi fino a 255 ms. La routine salva il contenuto del registro D0 nello stack hardware, in modo da poterlo successivamente ripristinare. Potremmo servirci di una delle tecniche generali per il passaggio dei parametri, descritte nel Capitolo 11, in modo da scrivere una subroutine completamente "trasparente", che non modifichi nessun registro o flag. In tal caso, dovremmo includere, nella valutazione del tempo, quelle istruzioni che trasferiscono i parametri, salvano e ripristinano i registri e modificano opportunamente l'indirizzo di ritorno.

Esempio: Una subroutine di Ritardo

Scopo: Questa subroutine genera un ritardo di 1 millisecondo moltiplicato per il contenuto a 16 bit del registro dati D0.

Diagramma di Flusso 12-1

Il valore di MSCNT dipende dalla frequenza alla quale la CPU esegue le istruzioni.

Programma 12-1

```
00004100:                PROGRAM  EQU      $4100
                         ;
0000031D:                MSCNT    EQU      797               CONT. PER UN RITARDO DI 1 MS
                         ;
                                  ORG      PROGRAM

00004100: 2F01           PGM12_1  MOVE.L   D1,-(A7)          SALVA IL VALORE DEL REGISTRO
00004102: 5340                    SUBQ.W   #1,D0             DECREMENTA PER L'ISTR. DBRA
00004104: 323C 031D      MSTIME   MOVE.W   #MSCNT,D1         INIZIALIZZA CONT.A 1 MS
00004108: 51C9 FFFE      MSLOOP   DBRA     D1,MSLOOP         RITARDO DI 1 MS
0000410C: 51C8 FFF6               DBRA     D0,MSTIME         CONT. NUMERO DI MILLISECONDI
00004110: 221F                    MOVE.L   (A7)+,D1          RIPRISTINA REGISTRO
                         ;
00004112: 4E75                    RTS

                                  END      PGM12_1
```

Valutazione del Tempo:

Istruzione		Numero di esecuzioni
MOVE.L	D1,-(A7)	1
SUB.Q	#1,D0	1
MOVE.W	#MSCNT,D1	(D0)
DBRA	D1,MSLOOP	(D0)*MSCNT + 1
DBRA	D0,MSTIME	(D0)
MOVE.L	(A7) + ,D1	1

Il tempo totale trascorso è uguale a (D0) moltiplicato 1 MS. Se la memoria sta funzionando alla velocità massima e *senza stati di attesa,* le istruzioni richiedono il seguente numero di cicli di clock (cfr. l'Appendice B).

Istruzione		Numero di Cicli
MOVE.L	D1,-(A7)	16
SUB.Q	#1,D0	4
MOVE.W	#MSCNT,D1	8
DBRA	D1,MSLOOP	10(14 l'ultima volta)
DBRA	D0,MSTIME	10(14 l'ultima volta)
MOVE.L	(A7) + ,D1	12

Perciò, il programma richiederà

(DO) X (8 + 10 X MSCNT + 14 + 10) + 16 + 4 + 4 + 12 cicli di clock

ovvero:

(DO) X (32 + 10 X MSCNT) + 36 cicli di clock

Trentadue è la somma dei cicli richiesti da MOVE.W #MSCNT,D1, da DBRA D0,MSTIME e dei 14 cicli necessari all'ultima esecuzione di DBRA D1,MSLOOP. Dieci è il numero dei cicli dell'istruzione DBRA D1,MSLOOP. Infine, trentasei è la somma

dei cicli delle due istruzioni MOVE.L, dell'istruzione SUBQ.W e dei quattro cicli addizionali richiesti dall'ultima esecuzione di DBRA D0,MSTIME.

Così, per ottenere un intervallo pari ad 1 millisecondo

$$32 + 10 \times MSCNT = N_c$$

dove N_c è il numero dei cicli di clock per millisecondo. Ad una frequenza di clock di 8MHz per l'MC68000, $N_c = 8000$, perciò

$$10 \times MSCNT = 7968$$
$$MSCNT = 796.8$$

Se MSCNT è 796_{10}, il ritardo sarà di 1 millisecondo meno 1 microsecondo (8 cicli). Se MSCNT è 797_{10}, il ritardo è di un millisecondo più 0,250 microsecondi. Un errore di 1 microsecondo rappresenta, in un intervallo di 1 millisecondo, un errore pari allo 0,1%. Otterremo la percentuale d'errore più piccola prendendo 797_{10} come valore di MSCNT. Per eliminare il piccolo errore che si verifica nel caso in cui MSCNT=796, possiamo inserire due istruzioni NOP all'interno del ciclo, nel modo seguente:

```
MSTIME      MOVE.W      OMSCNT,D1
MSLOOP      DBRA        D1,MSLOOP
            NOP
            NOP
            DBRA        D0,MSTIME
```

Dal momento che ciascuna NOP occupa quattro cicli di clock, il microsecondo di errore prodottosi quando MSCNT è uguale a 796_{10} verrà ora eliminato. Prima di aggiungere queste due NOP, assicuratevi che il segnale di clock non provochi degli errori più grossi. Ad esempio, se il clock è preciso allo 0,2%, l'eliminazione di errori più piccoli in un loop di ritardo non darà dei risultati apprezzabili.

Anche con l'aggiunta delle due istruzioni NOP e con un clock esatto, il nostro programma non ha ancora raggiunto una precisione del 100%! I 36 cicli posti al di fuori di entrambi i loop si verificano indipendentemente dal valore contenuto nel registro D0. Queste istruzioni di inizializzazione potrebbero essere eliminate, se non fosse necessario salvare il valore di D1 e se D0 fosse inizializzato con il valore "count-1". Tuttavia, prima di apportare delle modifiche in questo senso, non dimenticate che anche le istruzioni di chiamata e di ritorno, al pari delle altre, contribuiscono alla durata del ritardo.

DISPOSITIVI LOGICI E FISICI[6]

Un obiettivo che dobbiamo porci quando realizziamo delle routine di I/O è quello di renderle indipendenti da un particolare tipo di

hardware. Queste routine trasferiranno, allora, dei dati da e verso dispositivi di I/O utilizzando gli indirizzi forniti loro sotto forma di parametri. Il dispositivo di I/O cui si può accedere tramite una particolare interfaccia viene chiamato *dispositivo fisico*. Il sistema operativo o il programma supervisore devono stabilire l'esatta corrispondenza fra i dispositivi logici e quelli fisici; devono, cioè, definire gli effettivi indirizzi fisici di I/O e le caratteristiche di cui dovremo tener conto nelle varie routine.

Dispositivo Fisico di I/O

Vantaggi della distinzione tra dispositivi logici e dispositivi fisici

Ecco i vantaggi di questo metodo:

1. **Il sistema operativo consente all'utente di cambiare il dispositivo di I/O,** sostituendo, ad esempio, le normali periferiche con un pannello di collaudo o un'interfaccia per un sistema di sviluppo. Questo si rivela particolarmente utile, oltre che per la manutenzione del sistema, anche quando si tratterà di collaudarlo per eliminare eventuali errori. Inoltre, l'utente finale ha la possibilità di cambiare in ogni momento la periferica in base alle sue necessità; è il tipico caso di un'uscita intermedia destinata ad un terminale video e di un'uscita finale diretta ad una stampante, oppure quello di avere alcuni input provenienti da una linea di comunicazione a distanza invece che dalla tastiera locale (da cui gli input provengono normalmente).
2. **Una stessa routine di I/O è in grado di gestire parecchi dispositivi identici o con caratteristiche simili.** È sufficiente che Il sistema operativo o l'utente forniscano l'indirizzo di una particolare telescrivente, di un terminale RS-232, di una stampante o di un'altra periferica.
3. **È estremamente semplice apportare delle modifiche alla configurazione di I/O;** basta solamente modificare la corrispondenza tra dispositivi logici e dispositivi fisici. Nel caso del microprocessore MC68000, le routine di I/O possono mantenersi indipendenti da determinati indirizzi fisici se utilizzano l'indirizzamento indiretto a registro indirizzi con indice, che consentirà all'utente di accedere ad un dispositivo fisico mediante una tabella.

TABELLA DEI DISPOSITIVI DI I/O

Selezione di una particolare periferica

Se il sistema ha in memoria una tabella con gli indirizzi delle periferiche (che inizia, ad esempio, all'indirizzo IODEV), tutto quello di cui una routine di I/O ha bisogno è un indice per la lettura di questa tabella. In questo modo, può selezionare una periferica servendosi dell'indirizzamento indiretto a registro indirizzi con indice. Se, ad esempio, l'indirizzo di un particolare dispositivo di I/O occupa nella tabella la posizione contrassegnata con DEV, sarà necessario un programma di questo tipo per effettuare il calcolo dell'indice e caricare l'indirizzo di base di questa periferica nel registro indirizzi A0:

```
MOVEQ.W   #DEV,DO        PRENDI IL NUMERO DEL DISPOSITIVO
LSL.W     #2,DO          MOLTIPLICA PER 4 (INDIRIZZI DI 4 BYTE)
MOVEA.L   #IODEV,A0      PRENDI L'INDIRIZZO DELLA TABELLA DI
                         I/O
LEA       (A0,D0.W),A0   PRENDI L'INDIRIZZO DEL DISPOSITIVO
```

A questo punto, il programma provvederà a trasferire i dati servendosi di un'istruzione

```
MOVE.B   DATA,0(A0)   INVIA UN DATO AL DISP. LOGICO DI I/O
```

oppure

```
MOVE.B   0(A0),DATA   PRENDI UN DATO DA UN DISP. LOGICO DI
                      I/O
```

Adottando questo metodo un'unica routine è sufficiente per trasferire dati da e verso tutta una serie di dispositivi di I/O, con il programma principale che si limita semplicemente a fornire l'indice per la consultazione della tabella. L'indirizzamento assoluto non consente questo tipo di flessibilità e le routine che ne fanno uso devono inevitabilmente riferirsi sempre agli stessi indirizzi fisici.

CHIP DI INPUT/OUTPUT PER L'MC68000

La maggior parte delle routine di I/O dell'MC68000 sono basate su dei chip d'interfacciamento LSI. Questi dispositivi contengono latch, buffer, flip-flop ed altri circuiti logici necessari per l'handshake ed altre semplici tecniche d'interfacciamento. Racchiudono anche molte connessioni logiche e di queste possiamo selezionarne alcune in particolare, variando il contenuto di determinati registri programmabili. In questo modo il programmatore ha a sua disposizione l'equivalente di un "Manuale del Progettista di Circuiti". La fase di inizializzazione del programma mette nei registri i valori necessari alla selezione delle connessioni logiche desiderate. Le routine di input/output basate su chip d'interfacciamento programmabili LSI possono gestire molte applicazioni diverse ed eventuali cambiamenti sono possibili via software, senza la necessità di una modifica dei collegamenti fisici.

Con l'MC68000 vengono normalmente impiegati i seguenti chip d'interfacciamento:

1. **Il 6821 Peripheral Interface Adapter** (Adattatore d'Interfaccia Periferica), che descriveremo nel capitolo seguente. Contiene due porte ad 8 bit e quattro linee di controllo.
2. **Il 6850 Asynchronous Communications Interface Adapter** (Adattatore d'Interfacciamento Asincrono per Comunicazioni), che converte i dati dal formato parallelo ad 8 bit a quello seriale richiesto dalla maggior parte delle applicazioni nel campo delle comunicazioni. Parleremo dell'ACIA 6850 nel Capitolo 14.

BIBLIOGRAFIA

1. J. Barnes and V. Gregory. "Use Microprocessors to Enhance Performance with Noisy Data," *END*, August 20, 1976, pp. 71-72.
2. S.V. Alekar. "M6800 Program Performs Cyclic Redundancy Checks." *Electronics,* December 6, 1979, p. 167.
   J.E. McNamara, *Technical Aspects of Data Communications.* Maynard, Mass.: Digital Equipment Corporation, 1977, Chapter 13.
   R. Swanson. "Understanding Cyclic Redundancy Codes," *Computer Design*, November 1975, pp. 93-99.
   J. Wong et al. "Software Error Checking Procedures for Data Communicatio,ns Protocols," *Computer Design,* February 1979, pp. 122-25.
3. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 - Some Real Microprocessors.* Berkeley: Osborne/McGraw-Hill, 1978, pp. 9-106 through 9-123.
4. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 - Some Real Microprocessors.* Berkeley: Osborne/McGraw-Hill, 1978, pp. 9-78 through 9-106.
5. A. Osborne et al. *An Introduction to Microcomputers: Volume 2 - Some Real Microprocessors.* Berkeley: Osborne/McGraw-Hill, 1978, pp. 9-124 through 9-130.
6. C.W. Gear. *Computer Organization and Programming,* 3rd ed. New York: McGraw-Hill, 1980, Chapter 6.

CAPITOLO 13

L'USO DEL PIA 6821 (Peripheral Interface Adapter)

In questo capitolo parleremo del PIA 6821[1,2], un dispositivo che consente vari tipi di I/O parallelo, analizzando nei dettagli le modalità di programmazione e fornendo parecchi esempi di routine di I/O.

REGISTRI E LINEE DI CONTROLLO

Componenti del PIA

La Figura 13-1 mostra il diagramma a blocchi di un PIA. Si tratta di un dispositivo con due porte ad 8 bit pressochè identiche: la A che viene normalmente usata come ingresso e la B che, di solito, è destinata all'output. Ogni porta dispone di:

- **Un registro dati o periferico,** che contiene il dato in entrata o quello in uscita. Questo registro dispone di latch in fase di output, ma non in fase di input.
- **Un registro direzione dati.** I bit di questo registro stabiliscono se i corrispondenti bit (e pins) del registro dati sono degli ingressi (0) o delle uscite (1).
- **Un registro di controllo,** con i segnali di stato necessari per la procedura di handshake ed altri bit che servono a selezionare particolari connessioni logiche all'interno del PIA.
- **Due linee di controllo,** configurabili mediante i registri di controllo ed utilizzate per i segnali di handshake mostrati nelle Figure 12-5 e 12-6.

La funzione dei bit del registro direzione dati e del registro di controllo dipende esclusivamente dall'hardware. Un programmatore in linguaggio assembly non può far altro che consultare le tabelle che riportiamo nelle pagine seguenti ed, eventualmente, cercare di memorizzarle.(Le tabelle in questione vanno dalla 13.2 alla 13.6).

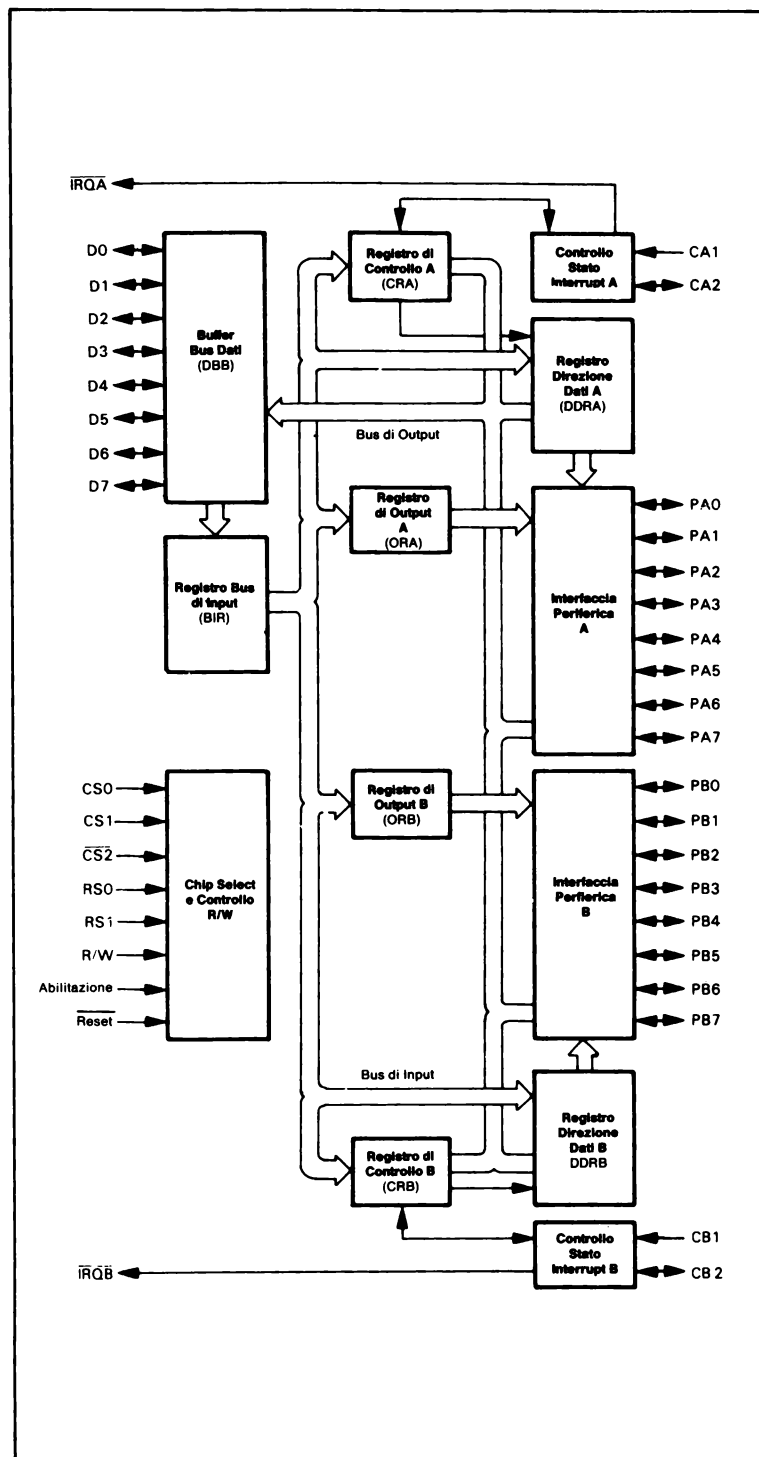



Figura 13-1. Diagramma a Blocchi del 6821 Peripheral Interface Adapter.

Indirizzi

Ogni PIA occupa quattro indirizzi di memoria. Le linee RS (Register Select) permettono di selezionare uno dei quattro registri (cfr. Tabella 13-1), ma, dal momento che complessivamente i registri sono sei (due periferici, due direzione dati e due di controllo), **sarà necessario un altro bit per poterli indirizzare tutti. Viene utilizzato, a questo scopo, il bit 2 dei registri di controllo, che seleziona il corrispondente registro direzione dati (se il suo valore è 0) oppure il registro periferico (se il suo valore è 1).** Questo significa che:

Indirizzamento dei registri direzione dati

1. Per poter cambiare il tipo del registro utilizzato (ad esempio passare da un registro direzione dati a un registro periferico) un programma deve cambiare il bit 2 nel registro di controllo.
2. Il programmatore deve conoscere il contenuto del registro di controllo per sapere quale registro sta indirizzando. Il segnale di RESET azzera il registro di controllo e, perciò, seleziona il registro direzione dati.

La Tabella 13-1 mostra, anche, come indirizzare i registri di un PIA. Normalmente, le linee RS0 e RS1 sono collegate alle linee indirizzi A1 e A2, mentre le linee dati corrispondono agli otto bit, superiori o inferiori, del bus dati. Questo significa che i registri del PIA occupano locazioni di memoria alternate, cioè soltanto indirizzi dispari oppure solo indirizzi pari. (In proposito vi forniremo maggiori dettagli nelle pagine seguenti). Perciò, caricando in uno dei registri indirizzi dell'MC68000 l'indirizzo corrispondente al registro dati (periferico) A, potremo indicare gli altri registri mediante gli offset che appaiono nella colonna più a destra della Tabella 13-1.

I Registri di Controllo del PIA

La Tabella 13-2 mostra come sono organizzati i registri di controllo del PIA. Ogni bit ha un significato particolare:

Bit 7: Viene posto a 1 da transizioni sulla linea di controllo 1 ed azzerato dalla lettura del registro periferico (o dati)

Bit 6: Analogo al bit 7, tranne per il fatto di essere posto a 1 da transizioni sulla linea di controllo 2

Bit 5: Stabilisce la configurazione della linea di controllo 2: 0 per l'input, 1 per l'output

Bit 4: Se la linea di controllo 2 è configurata come ingresso, serve a stabilire se il bit 6 deve essere posto a 1 da transizioni alto-basso (0) o basso-alto (1), sulla linea di controllo 2
Se la linea di controllo 2 è configurata come uscita, lo stato di questo bit determina la presenza sulla linea di controllo 2 di un impulso (0) oppure di un livello (1)

Tabella 13-.1 Indirizzamento dei Registri Interni del PIA 6821

Linee Indirizzi		Bit del Registro di Controllo		Registro Selezionato	Offset di Indirizzamento (Registro Indice o Puntatore di Stack) = Indirizzo del Registro (Dati) Periferico A
RS1	RS0	CRA-2	CRB-2		
0	0	1	X	Registro Periferico A	0
0	0	0	X	Registro Direzione Dati A	0
0	1	X	X	Registro di Controllo A	1
1	0	X	1	Registro Periferico B	2
1	0	X	0	Registro Direzione Dati B	2
1	1	X	X	Registro di Controllo B	3

X = 0 oppure 1

Tabella 13-2. Organizzazione dei Registri di Controllo del PIA

	7	6	5	4	3	2	1	0
CRA	IRQA1	IRQA 2	Controllo CA2			Accesso a DDRA	Controllo CA1	
	7	6	5	4	3	2	1	0
CRB	IRQB1	IRQB2	Controllo CB2			Accesso a DDRP	Controllo CB1	

Tabella 13-3. Controllo degli Input di interrupt CA1 e CA2 del PIA 6821

CRA-1 (CRB-1)	CRA-0 (CRB-0)	Input di Interrupt CA1 (CB1)	Flag di Interrupt CRA-7 (CRB-7)	Richiesta di Interrupt MPU $\overline{IRQA}$ ($\overline{IRQB}$)
0	0	↓ Attivo	Alto su ↓ di CA1 (CB1)	Disabilitatto -$\overline{IRQ}$ rimane alto
0	1	↓ Attivo	Alto su ↓ di CA1 (CB1)	Diventa basso quando il flag di Interrupt (bit CRA-7 o CRB-7) diventa alto
1	0	↑ Attivo	Alto su ↑ di CA1 (CB1)	Disabilitato -$\overline{IRQ}$ rimane alto
1	1	↑ Attivo	Alto su ↑ di CA1 (CB1)	Diventa basso quando il flag di Interrupt (bit CRA-7 o CRB-7) diventa alto

Note:

1: ↑ indica transizione positiva (basso-alto)
2: ↓ indica transizione negativa (alto-basso)
3: Il flag di Interrupt (bit CRA-7) è azzerato da una lettura del Registro Dati A da parte della MPU e CRB-7 è azzerato da una lettura del Registro Dati B da parte della MPU
4: Se CRA-0 (CRB-0) è basso quando si verifica un interrutp (interrupt disabilitato) e poi diventa alto, allora $\overline{IRQA}$ ($\overline{IRQB}$) si verifica dopo che in CRA-0 (CRB-0) è stato scritto un "1".

Bit 3: Con la linea di controllo 2 configurata come ingresso, un valore 1 abilita l'interrupt corrispondente al bit 6
Con la linea di controllo 2 configurata come uscita, stabilisce le condizioni finali di un impulso (0 = la fase di riconoscimento dell'handshake continua fino alla successiva transizione sulla linea di controllo 1; 1 = un breve strobe che ha la durata di un ciclo di clock) oppure il valore di un livello

Bit 2: Permette di selezionare il registro direzione dati (0) oppure il registro dati (1)

Bit 1: Stabilisce se il bit 7 è posto a 1 da transizioni alto-basso (0) o basso-alto (1) sulla linea di controllo 1

Bit 0: Se vale 1, abilita l'interrupt corrispondente al bit 7 del registro di controllo

Le Tabelle da 13-3 a 13-6 forniscono maggiori dettagli riguardo alla funzione dei singoli bit. L'impulso "E" corrisponde all'impulso di clock.

Tabella 13-4. Controllo delle linee di Interrupt CA2 e CB2 del PIA 6821 (CRA5 (CRB5) è Basso)

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRB-3)	Input di Interrupt CA2 (CB2)	Flag di Interrupt CRA-6 (CRB-6)	Richiesta di Interrupt MPU $\overline{IRQA}$ ($\overline{IRQB}$)
0	0	0	↓ Attivo	Alto su ↓ di CA2 (CB2)	Disabilitato - $\overline{IRQ}$ rimane alto
0	0	1	↓ Attivo	Alto su ↓ di CA2 (CB2)	Diventa basso quando il flag di Interrupt (CRA-6 o CRB-6) diventa alto
0	1	0	↑ Attivo	Alto su ↑ di CA2	Disabilitato -$\overline{IRQ}$ rimane alto
0	1	1	↑ Attivo	Alto su ↑ di CA2	Diventa basso quando il flag di Interrupt
				(CB2)	(CRA-6 o CRB-6) diventa alto

Note:

1: ↑ indicaa transizione positiva (basso-alto)
2: ↓ indica transizione negativa (alto-basso)
3: Il flag di Interrupt (bit CRA-6) è azzerato da una lettura del Registro. Dati A da parte della MPU e CRB-6 è azzerato da una lettura del Registro. Dati B da parte della MPU
4: Se CRA-3 (CRB-3) è basso quando si verifica un interrutp (interrupt disabilitato) e poi diventa alto, allora $\overline{IRQA}$ ($\overline{IRQB}$) si verifica dopo che in CRA-3 (CRB-3) è stato scritto un "1"

Tabella 13-5. Controllo della linea di Output CB2 del PIA 6821 :CRB5 è Alto)

CRB-5	CRB-4	CRB-3	CB2		Modo
			Azzerato	Posto a 1	
1	0	0	Basso sulla transizione positiva del primo impulso E successivo ad un'operazione di scrittura della MPU nel Registro Dati "B"	Alto quando il flay di interrupt (bit CRB-7) è posto a 1 da una transizione attiva del segnale CB1	Riconoscimento Output Automatico
1	0	1	Basso sulla transizione positiva del primo impulso E successivo ad un'operazione di scrittura della MPU nel Registro Dati "B"	Alto sul fronte positivo del primo impulso "E" successivo ad un impulso "E" verificatasi quando la parte non era selezionata	Strobe (Scrittura) Output Automatico
1	1	0	Basso quando CRB-3 diventa basso con risultato di una operazione di scrittura della MPU nel Registro di Controllo "B"	Sempre basso finché CRB-3 è basso. Diventerà alto in seguito ad una operazione di scrittura della MPU nel Registro di Controllo "B" che cambi CRB-3 a "1"	Output Manuale (Basso)
1	1	1	Sempre alto finché CRB-3 è alto. Sarà azzerato quando una operazione di scrittura della MPU nel Registro di Controllo "B" pone CRB-3 a "zero"	Alto quando CRB-3 diventa alto in seguito ad un'operazione di scrittura nel Registro di Controllo "B".	Output Manuale (Alto)

Tabella 13-6. Controllo della Linea di Ouput CA2 del PIA 6821 (CRA5 è Alto)

CRA-5	CRA-4	CRA-3	CA2 Azzerato	CA2 Posto a 1	Modo
1	0	0	Basso sulla transizione negativa di E dopo un'operazione di lettura della MPU dal Registro Dati "A"	Alto quando il flag di interrupt (CRA-7) è posto a 1 da transiziofne attiva del segnale CA1	Riconoscimento (Acknowledge) Input Automatico
1	0	1	Basso sulla transizione negativa di E dopo un'operazione di lettura della MPU dal Registro Dati "A"	Alto sul fronte negativo del primo impulso "E" che si verifica quando la parte non è selezionata	Strobe (Lettura) Input Automatico
1	1	0	Basso quando CRA-3 diventa basso con risultato di una operazione di scrittura della MPU nel Registro di Controllo "A"	Sempre basso finché CRA-3 è basso. Diventerà alto in seguito ad una operazione di scrittura della MPU nel Registro di Controllo "A" che cambi CRA-3 a "1"	Output Manuale (Basso)
1	1	1	Sempre alto finché CRA-3 è alto. Sarà azzerato quando una operazione di scrittura della MPU nel Registro di Controllo "A" pone CRA-3 a "zero"	Alto quando CRA-3 diventa alto in seguito ad un'operazione di scrittura della MPU nel Registro di Controllo "A".	Output Manuale (Alto)

COME INIZIALIZZARE UN PIA

Le modalità di funzionamento del PIA devono essere definite durante la fase di inizializzazione del sistema. Un PIA contiene, come i processori, un gran numero di connessioni logiche. Siamo in grado di attivare solo quelle che ci interessano caricando determinati valori sui registri di controllo e direzione dati. Un pò la stessa cosa che accade quando viene caricato un dato nel registro istruzioni della CPU. La differenza sta nel fatto che il PIA contiene un numero molto inferiore di connessioni rispetto ad un microprocessore e, normalmente, una volta che queste sono state attivate non vengono più modificate (o lo sono molto raramente).

Queste sono le fasi necessarie per definire il funzionamento di un PIA:

1. **Indirizzare i registri direzione dati,** azzerando il bit 2 di ogni registro di controllo, per stabilire quali pin dovranno servire come input e quali come ouput. Dato che un segnale di RESET azzera l'intero registro di controllo, questa fase non è necessaria al momento dell'attivazione iniziale del sistema.
2. **Stabilire quali pin saranno utilizzati come ingressi e quali come uscite,** caricando le opportune combinazioni di 0 (input) e di 1 (output) nei registri direzione dati.
3. **Stabilire come dovranno operare le linee di stato e di controllo,** assegnando gli opportuni valori ai bit 0, 1, 3, 4 e 5 dei registri di controllo. Indirizzare i registri dati, mettendo a 1 il bit 2 di ciascun registro di controllo.

Un registro direzione dati può essere indirizzato in questo modo:

```
CLR.B PIACA  AZZERA IL REG. DI CONTROLLO DEL PIA (LATO A)
```

oppure

```
MOVE.B  #$FB,D0
AND.B   D0,PIACA   SELEZIONA IL REGISTRO DIREZIONE DATI
```

oppure

```
BCLR.B  #2,PIACA   SELEZIONA IL REGISTRO DIREZIONE DATI
```

Il secondo ed il terzo metodo hanno validità più generale, dal momento che non cambiano nessuno degli altri bit del registro di controllo, ma agiscono selettivamente sul bit 2.

Una volta che abbiamo selezionato il registro direzione dati, possiamo definire una qualsiasi combinazione di ingressi e di uscite, memorizzando in questo registro una opportuna sequenza di 0 e 1. Ecco alcuni esempi:

```
CLR.B   PIADDA          TUTTE LE LINEE DATI SONO INGRESSI

MOVE.B  #$FF,PIADDA     TUTTE LE LINEE DATI SONO USCITE

MOVE.B  #$F0,PIADDA     LE LINEE DATI 4-7 SONO USCITE, LE 0-3 IN-
                        GRESSI
```

La terza fase è chiaramente la più complessa, poichè comporta la selezione delle connessioni logiche attive che determineranno le modalità di funzionamento del PIA.

Alcuni aspetti da tener presenti sono:

1. **Non è possibile cambiare il valore dei bit 6 e 7 di un registro di controllo con un'operazione di scrittura.** Solo delle transizioni sulle linee di controllo mettono ad 1 questi bit e li azzera soltanto la lettura dei corrispondenti registri dati.
2. **Per selezionare il registro periferico, permettendo il trasferimento di dati da e verso il mondo esterno, bisogna porre a 1 il bit 2 di ciascun registro di controllo.** Finchè il valore di questo bit rimane zero, la CPU può accedere soltanto al corrispondente registro direzione dati e non può effettuare nessun trasferimento da e verso i pin di I/O, che è possibile solo attraverso il registro dati.
3. **Il bit 1 del registro di controllo stabilisce quale dei due fronti di un impulso, presente sulla linea di controllo 1, servirà per mettere a 1 il bit 7.** Se il valore di questo bit è 0, sarà un fronte di discesa, cioè una una transizione alto-basso, a porre ad uno il bit 7; se il valore di questo bit è 1, allora a svolgere questa funzione sarà una transizione basso-alto (fronte di salita). Il bit 4 ha una funzione analoga relativamente alla linea di controllo 2, quando questa è configurata come ingresso.

4. **Per abilitare l'interrupt per la linea di controllo 1** bisogna porre a uno il bit 0 del registro di controllo. Degli interrupt parleremo in seguito, nel Capitolo 15. Il bit 3 svolge la stessa funzione relativamente alla linea di controllo 2, quando questa è configurata come ingresso.
5. **Il bit 5 stabilisce se la linea di controllo 2 deve essere un'uscita (1) oppure un ingresso (0).** I bit 3 e 4 definiscono le caratteristiche di funzionamento della linea di controllo 2. In caso di impulso o strobe automatico, le porte A e B si comportano in modo diverso; la porta A produce un impulso su CA2, solo dopo che il processore ha letto il registro dati A, mentre la porta B produce un impulso su CB2, solo dopo che il processore ha scritto nel registro dati B.
6. **È indispensabile definire le modalità operative delle porte di tutti i PIA presenti in un sistema.** Ogni porta dispone di un proprio registro di controllo, di un registro direzione dati e di un registro dati.

MODI OPERATIVI DEL PIA

Con le linee CA2 o CB2 utilizzate per inviare segnali di controllo in uscita sono possibili vari modi operativi.

Modi automatici

Parleremo di modi *automatici* quando il PIA produce automaticamente un impulso su CA2 dopo un'operazione di input o su CB2 dopo un'operazione di output. È il PIA, infatti, che genera l'impulso, senza nessun intervento da parte della CPU. Il programmatore non ha la possibilità di modificarne la lunghezza o la polarità.

Modo manuale

Parleremo di *modo manuale* quando è il bit 3 del registro di controllo del PIA a determinare il livello del segnale sulla linea di controllo 2. È indispensabile, infatti, l'intervento della CPU per azzerare o porre a uno questo bit. Il PIA non fa niente automaticamente e bisogna far ricorso a delle istruzioni aggiuntive, ma in compenso il programmatore può controllare la lunghezza e la polarità degli impulsi.

Nei due modi automatici la linea di controllo 2 ha le seguenti funzioni:

Quando l'impulso automatico continua fino alla succesiva transizione attiva sulla linea di controllo 1, la linea di controllo 2 serve come *riconoscimento* (acknowledgement). La parte attiva dell'impulso (il periodo basso) significa che la CPU ha completato quella fase dell'operazione di I/O che è di sua competenza; perciò la periferica può iniziare la fase successiva, inviando un dato (nel caso che si tratti di un'operazione di input) o indicando la sua disponibilità a riceverne uno (output).

Quando l'impulso ha la durata di un ciclo di clock la linea di controllo 2 svolge una funzione di strobe, indicando che la CPU ha eseguito un'operazione di I/O.

Come Selezionare i Modi Operativi

1. **Una semplice porta di Input senza linee di controllo** (necessaria, ad esempio, nel caso di una serie di interruttori):

```
CLR.B    PIACA        SELEZIONA IL REGISTRO DIREZIONE
                      DATI
CLR.B    PIADDA       TUTTE LE LINEE DATI SONO INGRESSI

BSET.B   #2,PIACA     INDIRIZZA IL REGISTRO DATI
```

Funzionamento della porta di input

Dopo aver azzerato il bit 2 del registro di controllo per poter accedere al registro direzione dati, il programma configura come ingressi tutte le linee dati, mettendo degli zeri in tutti i bit di questo registro (e nella stessa porta di input). La stessa sequenza di istruzioni è valida anche nel caso in cui sia utilizzata una transizione alto-basso (fronte di discesa) sulla linea di controllo 1 per indicare la disponibilità di un nuovo dato (DATA READY) o la disponibilità della periferica (PERIPHERAL READY).

2. **Una semplice porta di output senza linee di controllo** (necessaria, ad esempio, per controllare un gruppo di LED):

```
CLR.B    PIACA        SELEZIONA IL REGISTRO DIREZIONE
                      DATI

MOVE.B   #$FF,PIADDA  TUTTE LE LINEE DATI SONO USCITE

BSET.B   #2,PIACA     INDIRIZZA IL REGISTRO DATI
```

La sola differenza rispetto all'esempio precedente sta nel fatto che il programma configura come uscite tutte le linee dati, memorizzando degli uno in tutti i bit del registro direzione dati.

3. **Una porta di input con una transizione basso-alto (positiva) sulla linea di controllo 1, che indica la disponibilità di un dato (DATA READY).**

```
CLR.B    PIACA        SELEZIONA IL REGISTRO DIREZIONE
                      DATI
CLR.B    PIADDA       TUTTE LE LINEE DATI SONO INGRESSI

MOVE.B   #$06,PIACA   DATA READY CON BASSO-ALTO
```

La sola differenza con l'Esempio 1 sta nel fatto che il programma mette ad uno il bit 1 del registro di controllo, in modo che il bit 7 di questo stesso registro sia attivato dalle transizioni basso-alto che avvengono sulla linea di controllo 1. Questo modo operativo è adatto per la maggior parte delle tastiere codificate.

4. **Una porta di output che produce un breve strobe per indicare la disponibilità di un dato (DATA READY o OUTPUT READY).** Questo strobe potrebbe servire a multiplexare dei display o a segnalare la disponibilità di un dato ad una stampante.

```
CLR.B     PIACA          SELEZIONA IL REGISTRO DIREZIONE
                         DATI

MOVE.B    #$FF,PIADDA    TUTTE LE LINEE DATI SONO USCITE

MOVE.B    #$2C,PIACA     LA LINEA DI CONTR. 2 PRODUCE BREVE
                         STROBE
```

Le caratteristiche del segnale presente sulla linea di controllo 2 sono determinate nel modo seguente:
Bit 5 = 1 per configurare come uscita la linea di controllo 2
Bit 4 = 0 perchè deve trattarsi di un impulso e non di un livello.
Bit 3 = 1 perchè l'impulso deve avere una durata pari di un ciclo di clock.
Dopo ogni istruzione che scrive un dato nel registro dati B di un PIA la linea di controllo 2 diventa bassa per tutta la durata di un ciclo di clock. Ad esempio, l'istruzione

```
MOVE.B    DO,PIADB
```

invia un dato al registro dati (e di qui alla porta di output) e, allo stesso tempo, genera uno strobe sulla linea di controllo 2. Al contrario, la porta A di un PIA produce uno strobe soltanto dopo un'operazione di lettura. La sequenza

```
MOVE.B    DO,PIADA       SCRIVI UN DATO
MOVE.B    PIADA,DO       FALSA LETTURA: CAUSA STROBE IN U-
                         SCITA
```

invia un dato alla porta di output e, contemporaneamente, genera uno strobe. L'istruzione MOVE.B PIADA,DO è una "falsa lettura" e non ha altro effetto che quello di generare uno strobe (e di far trascorrere pochi cicli di clock). Lo stesso scopo si potrebbe raggiungere anche con altre istruzioni; provate a indicarne qualcuna.

5. **Una porta di input con uno strobe di handshake per il riconoscimento dell'input (INPUT ACKNOWLEDGE).** Il segnale di strobe diventa basso quando la CPU ha letto il dato presente nella porta ed è pronta ad accettarne un altro.

```
CLR.B     PIACA          SELEZIONA IL REGISTRO DIREZIONE
                         DATI
CLR.B     PIADDA         TUTTE LE LINEE DATI SONO INGRESSI
MOVE.B    #$24,PIACA     LINEA CONTROLLO 2: HANDSHAKE AC-
                         KNOWLEDGE
```

Il bit 5 del registro di controllo è uguale a 1 perchè la linea di controllo 2 deve essere un'uscita, il bit 4 = 0 perchè deve trattarsi di un impulso ed il bit 3 = 0 perchè il segnale di riconoscimento è un segnale attivo-basso che rimane tale fino alla successiva transizione attiva sulla linea di controllo 1. La porta funziona in questo modo:

a. Una transizione alto-basso sulla linea di controllo 1 indica che la periferica di input ha inviato un nuovo dato all'elaboratore. Il bit 7 del registro di controllo del PIA viene posto ad uno e la linea di controllo 2 diventa alta.
b. La CPU, esaminato il bit 7 del registro di controllo e rilevata la disponibilità di un nuovo dato, provvede a prelevarlo dal registro dati. Con questa operazione viene azzerato il bit 7 del registro di controllo e la linea di controllo 2 diventa bassa.
c. La periferica di input, esaminando la linea di controllo 2, ha la conferma che la CPU ha accettato il dato che le è stato inviato. Può, allora, ripetere la fase 2, avendo l'assoluta certezza che nessun dato andrà perso.

Una qualunque istruzione che legga il registro dati A del PIA è seguita automaticamente da un segnale di riconoscimento; ad esempio,

```
MOVE.B PIADA,D0
```

oltre a leggere un dato genera un segnale di riconoscimento. Nel caso della porta B, al contrario, si ha la comparsa di un segnale di riconoscimento solo dopo un'istruzione che scrive nel registro dati. La sequenza

```
MOVE.B PIADB,D0 LEGGI UN DATO
MOVE.B D0,PIADB FALSA SCRITTURA: CAUSA RICONOSCI-
                MENTO
```

legge un dato e, allo stesso tempo, provoca un riconoscimento. L'istruzione MOVE.B D0,PIADB è una "falsa scrittura"; non avendo altro effetto che quello di causare un riconoscimento (di far, cioè, diventare bassa la linea di controllo 2) e di usare alcuni cicli di clock. Si noti che, in questo caso, le istruzioni sono in ordine inverso rispetto all'Esempio 4. Questo modo operativo si presta ad essere utilizzato con molti terminali CRT che richiedono un handshake completo.

6. **Una porta di output con un livello zero in uscita** mediante latch (un output seriale latched o un output di livello con valore 0). Si tratta di un output seriale che può servire ad accendere o spegnere una periferica oppure a stabilirne le caratteristiche operative.

```
CLR.B    PIACA      SELEZIONA IL REGISTRO DIREZIONE
                    DATI
```

```
MOVE.B   #$FF,PIADDA  TUTTE LE LINEE DATI SONO USCITE

MOVE.B   #$34,PIACA   LINEA DI CONTR. 2: LATCHED OUTPUT
                      VALORE INIZIALE 0
```

Il bit 5 = 1 perchè la linea di controllo 2 deve essere un'uscita, il bit 4 = 1 per avere su di essa un livello (latched bit) ed il bit 3 = 0 perchè il valore del livello deve essere zero. In questo caso, il segnale presente sulla linea di controllo 2 non cambierà automaticamente il suo valore in seguito ad operazioni di lettura o scrittura del registro dati. L'unico modo per modificare questo segnale è quello di cambiare il valore del bit 3 del registro di controllo del PIA, cioè:

```
MOVE.B   #$08,D0
OR.B     D0,PIADA     L'OUTPUT SERIALE A UNO

oppure

BSET.B   #$2C,PIACA   L'OUTPUT SERIALE A UNO
MOVE.B   #$F3,D0
AND.B    D0,PIACA     L'OUTPUT SERIALE A ZERO

BCLR.B   #3,PIACA     L'OUTPUT SERIALE A ZERO
```

Ci possiamo servire di questo modo operativo per produrre strobe attivi-alti oppure per generare impulsi la cui durata potrà essere stabilita dall'interno del programma e non dipenderà dalle caratteristiche hardware del sistema.

L'USO DEL PIA PER IL TRASFERIMENTO DI DATI

Una volta definite le modalità operative, i registri dati del PIA possono essere considerati come delle locazioni di memoria qualsiasi. **Il modo più semplice per prelevare un dato da un dispositivo di input o inviarne uno ad un dispositivo di output è quello di servirsi di istruzioni di questo tipo:**

Istruzioni utilizzabili con il PIA

MOVE.B <ea>,Dn trasferisce un valore ad 8 bit dai pin di input specificati ad un registro dati.

MOVE.B Dn,<ea> trasferisce un valore ad 8 bit da un registro dati ai pin di output specificati.

Naturalmente, ci sono dei casi in cui le porte di input ed output non si comportano come delle normali locazioni di memoria. Come, ad esempio, quando si tenti di scrivere un dato in una porta di input o di leggerne uno da una porta di output. Fate attenzione ad operazioni

di questo tipo, soprattutto se la porta di input non è stabilizzata con dei latch o se la porta di output non dispone di un buffer.

Nelle operazioni di I/O possono essere impiegate anche altre istruzioni che trasferiscono dati da e verso la memoria. Ad esempio:

Clear mette degli zeri su un gruppo di output.
Le istruzioni per la **manipolazione dei bit** possono porre a uno, azzerare o controllare un singolo pin. Non è possibile, tuttavia, modificare i bit di un dato del PIA.
CMP modifica i flag come se i valori di un gruppo di pin di input fossero stati sottratti dal contenuto di un registro dati, da un valore immediato oppure da una locazione di memoria.

Occorre, comunque, tenere sempre presenti le limitazioni fisiche delle porte di I/O, soprattutto con le istruzioni di Shift, con quelle che complementano un valore e con alcune istruzioni per la manipolazione dei bit, in quanto si tratta di istruzioni che richiedono operazioni sia di lettura che di scrittura.

Importanza della documentazione nei programmi per il PIA

Non ci stancheremo mai di ricordarvi quanto sia importante documentare un programma in modo chiaro ed esauriente. Molto spesso può accadere che delle istruzioni, il cui significato non è subito evidente, nascondano operazioni di I/O particolarmente complesse. Perciò, è necessario indicare esattamente la loro funzione, altrimenti potremmo, ad esempio, essere tentati di togliere quelle false operazioni di lettura o scrittura che abbiamo descritto in precedenza, perchè ci sembrano inutili.

I Bit di Stato del PIA

Il Bit 7 del registro di controllo del PIA ha spesso la funzione di un bit di stato per indicare, ad esempio, la disponibilità di un nuovo dato o quella di una periferica. Il valore di questo bit può essere controllato mediante una di queste sequenze:

```
MOVE.B   PIACA,D0    IL FLAG DI PRONTO È 1?
BMI      DEVRDY1     ...SÌ, DISPOSITIVO PRONTO

TST.B    PIACA       IL FLAG DI PRONTO È 1?
BMI      DEVRDY2     ...SÌ, DISPOSITIVO PRONTO

BTST.B   #7,PIACA    IL FLAG DI PRONTO È 1?
BNE      DEVRDY3     ...SÌ, DISPOSITIVO PRONTO
```

Ricordate di non utilizzare istruzioni di Shift, poiché esse modificano il contenuto del registro di controllo. (Perchè?) Il programma seguente attenderà che il flag di Pronto (bit 7) diventi alto:

```
MOVE.B   PIACA,D0    IL FLAG DI PRONTO È 1?
BPL      WAITR       ...NO, ATTENDI
```

Quali modifiche sarebbero necessarie se volessimo esaminare il bit 6 invece del bit 7?

L'unico modo per azzerare il bit 7 (o il bit 6) è di leggere il registro dati. Se la risposta prevista nel caso che il bit sia 1 non contiene un'operazione di lettura, possiamo sempre servirci di una falsa lettura. Se la porta viene utilizzata come uscita, la sequenza

```
MOVE.B   D0,PIACA     INVIA UN DATO
MOVE.B   PIADA,D0     FALSA LETTURA: AZZERA IL FLAG DI PRON-
                      TO
```

assolverà questo compito. In questo caso l'operazione di falsa lettura può essere effettuata sull'una o sull'altra delle porte di un PIA. Per azzerare lo strobe senza cambiare nulla, ad eccezione dei flag, possiamo far ricorso all'istruzione Test. Vi consigliamo di fare molta attenzione quando la CPU non è pronta a ricevere un dato oppure non ha un dato da inviare all'uscita.

ESEMPI DI PROGRAMMAZIONE

13-1. Un interruttore a pulsante

Interfacceremo un interruttore a pulsante ad un microprocessore MC68000 mediante un PIA 6821. Si tratta di un interruttore meccanico: premendo il pulsante lo chiudiamo mettendo a massa il bit di input (cfr. Fig. 13-2).

Il PIA 6821 agisce come un buffer e non è necessario un latch, dal momento che l'interruttore rimane chiuso per molti cicli di clock. La pressione del pulsante mette a massa un bit del PIA. Una resistenza di pullup mantiene a uno il bit di ingresso, quando il pulsante non viene premuto.

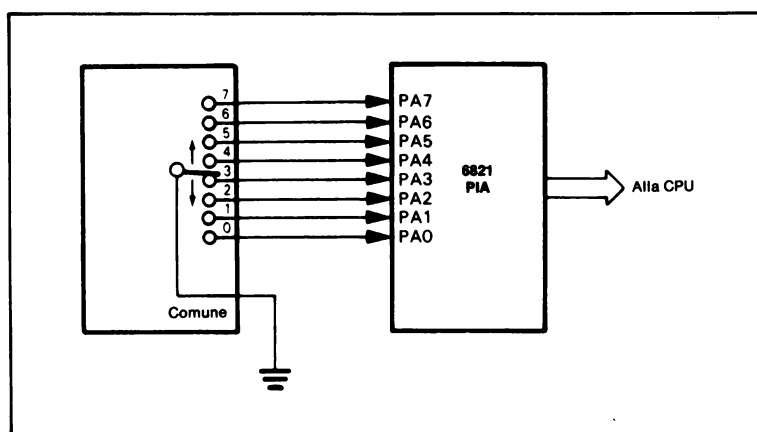



Figura 13-2. Un Circuito sa Pulsante.

Ci serviremo di questo circuito per realizzare due funzioni:

a. Azzerare o porre a uno una locazione di memoria in funzione dello stato del pulsante.
b Contare quante volte viene premuto il pulsante.

Funzione 13-1a. Determinare la Chiusura dell'Interruttore

Scopo: Se il pulsante non viene premuto porre ad uno la variabile BUTTON, alla locazione 6000; provvedere ad azzerarla quando viene premuto.

Casi Campione:

1. Aperto (pulsante non premuto)
   BUTTON = (6000) = 01

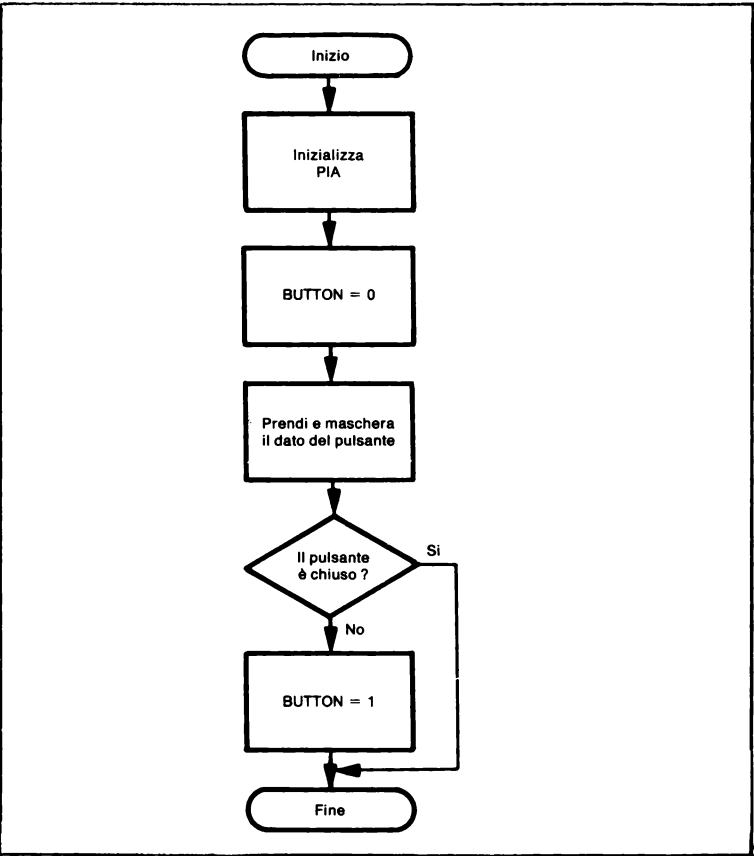
2. Chiuso (pulsante premuto)
   BUTTON = (6000) = 00

Programma 13-1a

```
00004000:                   PROGRAM  EQU     $4000
00006000:                   DATA     EQU     $6000
                            ;
0003FF40:                   PIADDA   EQU     $3FF40          REGISTRO DIREZIONE DATI A
0003FF40:                   PIADA    EQU     $3FF40          REGISTRO DATI A
0003FF44:                   PIACA    EQU     $3FF44          REGISTRO DI CONTROLLO A
                            ;
00000004:                   PIADS    EQU     $04             SELEZIONA REG. DATI DEL PIA
00000001:                   MASKBUT  EQU     $01             MASCHERA PER IL PULSANTE
                            ;
                                     ORG     DATA

00006000:                   BUTTON   DS.B    BUTTON          FLAG DI STATO
                            ;
                                     ORG     PROGRAM

00004000: 4239 0003
00004004: FF44              PGM13_1A CLR.B   PIACA           INDIRIZZA REG. DIREZIONE DATI
00004006: 4239 0003
0000400A: FF40                       CLR.B   PIADDA          TUTTE LE LINEE SONO INGRESSI
0000400C: 13FC 0004
00004010: 0003 FF44                  MOVE.B  #PIADS,PIACA    INDIRIZZA REGISTRO DATI
00004014: 4238 6000                  CLR.B   BUTTON          AZZERA FLAG
00004018: 1039 0003
0000401C: FF40                       MOVE.B  PIADA,D0        LEGGI POSIZIONE PULSANTE
0000401E: 0200 0001                  AND.B   #MASKBUT,D0     IL PULSANTE E' PREMUTO(0 LOGICO)?
00004022: 6704                       BEQ.S   DONE            SE SI' ALLORA VAI A DONE
00004024: 5238 6000                  ADDQ.B  #1,BUTTON       ...ALTRIMENTI BUTTON = 1
00004028: 4E75              DONE     RTS

                                     END     PGM13_1A
```

Gli indirizzi del registro di controllo A (PIACA), del registro direzione dati A (PIADDA) e del registro dati A (PIADA) dipendono dal modo in cui il PIA è stato collegato nel vostro elaboratore. In questo esempio non abbiamo utilizzato le linee di controllo.

Il valore di MASKBUT varia a seconda del bit con cui è collegato l'interruttore; ci sarà un 1 in quella posizione e degli zeri nelle altre.

Diagramma di Flusso 13-1a

Se il pulsante fosse collegato al bit 7 della porta di input del PIA potremmo controllare quando viene premuto servendoci di un'i-

Posizione del Pulsante (Numero Bit)	Maschera	
	Binaria	Esadecimale
0	00000001	01
1	00000010	02
2	00000100	04
3	00001000	08
4	00010000	10
5	00100000	20
6	01000000	40
7	10000000	80

struzione MOVE, TST o BTST ed esaminando il valore del flag di Segno (Negativo) o del flag di Zero:

```
MOVE.B   PIADA,D0    IL PULSANTE È PREMUTO(ZERO LOGICO)?
BPL      FATTO       ...SÌ, FATTO

TST.B    PIADA       IL PULSANTE È PREMUTO(ZERO LOGICO)?
BPL      FATTO       ...SÌ, FATTO

BTST.B   #7,PIADA    IL PULSANTE È PREMUTO(ZERO LOGICO)?
BEQ      FATTO       ...SÌ, FATTO
```

Con l'istruzione BTST possiamo controllare qualsiasi bit del registro dati.

Funzione 13-1b. Contare le Chiusure dell'Interruttore

Scopo: Contare quante volte viene premuto il pulsante, incrementando la variabile COUNT, alla locazione di memoria 6000, dopo ogni chiusura.

Caso Campione:

Premendo dieci volte il pulsante dopo l'inizio del programma, dovremmo ottenere:

COUNT = (6000) = 0A

Per stabilire con esattezza quante volte è stato premuto il pulsante dobbiamo essere sicuri che ad ogni chiusura corrisponda un'unica transizione. Purtroppo questo non accade con un interruttore meccanico, perchè i contatti rimbalzano avanti e indietro, prima di assumere la posizione finale. È un incoveniente che possiamo eliminare utilizzando un monostabile oppure tenendone conto nel programma.

In quest'ultimo caso, dobbiamo inserire un ciclo di ritardo ogni volta che viene premuto il pulsante. La durata di questo intervallo è indicata nelle specifiche dell'interruttore ed è in genere dell'ordine di alcuni millisecondi. In questa fase, il programma, non dovendo esaminare l'interruttore perchè altrimenti rischierebbe di scambiare i rimbalzi per chiusure successive, può dedicarsi ad altre attività oppure limitarsi semplicemente ad attendere.

Dopo aver eliminato questo fenomeno di rimbalzo, bisogna attendere che il pulsante venga rilasciato, prima di esaminarlo nuovamente. Inoltre, è necessario eliminare i rimbalzi anche al momento dell'apertura dell'interruttore. Questa procedura evita che una singola chiusura sia conteggiata due volte. Il programma che segue utilizza un ciclo di ritardo di 10 ms. Provate a variarne la durata o a toglierlo

completamente, per vedere cosa accade. Prima di eseguire questo programma alla locazione 4100 deve trovarsi la routine di ritardo.

Programma 13-1b

```
00004000:                 PROGRAM  EQU      $4000
00006000:                 DATA     EQU      $6000
00004100:                 DELAY    EQU      $4100         IND. DELLA ROUTINE DI RITARDO
                          ;
0003FF40:                 PIA      EQU      $3FF40        INDIRIZZO DI BASE DEL PIA
                          ;
00000000:                 PIADDA   EQU      $0            OFFSET REG. DIREZIONE DATI A
00000000:                 PIADA    EQU      $0            OFFSET REG. DATI A
00000004:                 PIACA    EQU      $4            OFFSET REG. DI CONTROLLO A
                          ;
00000000:                 A_DATDIR EQU      $00           NEL LATO A SONO TUTTI INGRESSI
00000004:                 A_CNTRL  EQU      04            SELEZIONA REGISTRO DATI
00000000:                 BUTBIT   EQU      $0            BIT DEL PULSANTE NEL PIA
0000000A:                 BOUNCE   EQU      10            TEMPO IN MS. PER ELIMINARE
                                                          I RIMBALZI
                          ;
                          ;
                                   ORG      DATA

00006000:                 COUNT    DS.B     1             CONT. PULSANTE PREMUTO
                          ;
                                   ORG      PROGRAM

00004000: 207C 0003
00004004: FF40            PGM13_1B MOVEA.L  #PIA,A0       PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004                CLR.B    PIACA(A0)     INIZIALIZZA UN LATO
0000400A: 117C 0000
0000400E: 0000                     MOVE.B   #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                     MOVE.B   #A_CNTRL,PIACA(A0)
                          ;
00004016: 4238 6000                CLR.B    COUNT         AZZERA CONTATORE
0000401A: 0828 0000
0000401E: 0000            CHKCLOSE BTST     #BUTBIT,PIADA(A0) IL PULSANTE E' PREMUTO?
00004020: 66F8                     BNE      CHKCLOSE      SE NO, ATTENDI CHE SIA PREMUTO
                          ;
00004022: 5238 6000                ADD.B    #1,COUNT      ...ALTRIMENTI INCREMENTA CONTATORE
00004026: 700A                     MOVEQ    #BOUNCE,D0    ATTENDI PER ELIMINARE RIMBALZI
00004028: 4EB8 4100                JSR      DELAY         IN CHIUSURA
0000402C: 0828 0000
00004030: 0000            CHKOPEN  BTST     #BUTBIT,PIADA(A0) IL PULSANTE E' ANCORA PREMUTO?
00004032: 67F8                     BEQ      CHKOPEN       SE SI', ATTENDI CHE SIA RILASCIATO
                          ;
00004034: 700A                     MOVEQ    #BOUNCE,D0    ATTENDI PER ELIMINARE RIMBALZI
00004036: 4EB8 4100                JSR      DELAY         IN APERTURA
0000403A: 60DE                     BRA      CHKCLOSE      ...ALTRIMENTI RICOMINCIA

                                   END      PGM13_1B
```

Diagramma di Flusso 13-1b

Spiegazione del programma 13-1b

L'uso del simbolo BOUNCE permette di cambiare contemporaneamente la durata di tutti i cicli di ritardo, senza doverli cercare all'interno del programma, ma variando soltanto il valore assegnato a questa label.

L'inizializzazione è leggermente diversa da quella del Programma 13-1a. Questa procedura, consistente nell'azzerare il registro di controllo, nel mettere un byte nel registro direzione dati e uno nel registro di controllo, possiamo utilizzarla sempre, assegnando eventualmente valori diversi alle variabili A_DATDIR e A_CNTR. È importante che la fase di inizializzazione sia facilmente modificabile, soprattutto nel caso delle routine di I/O, che sono soggette a frequenti cambiamenti in fase di collaudo.

Invece dell'indirizzamento assoluto lungo che abbiamo utilizzato nel programma precedente, questa volta abbiamo adottato quello indiretto a registro indirizzi con spostamento, che richiede un minor numero di byte ed accorcia il programma. È anche il solo indirizzamento possibile con MOVEP, un'istruzione molto utile nei programmi di I/O e di cui parleremo in seguito. Inoltre con l'indirizzamento assoluto lungo è necessario indicare gli indirizzi di tutti i registri di ogni PIA e, con un sistema che dispone di un gran numero di PIA, il compito risulta particolarmente gravoso. Nel caso dell'indirizzamento indiretto con spostamento, invece, è sufficiente specificare l'indirizzo di base di ciascun PIA e, solo una volta, i valori di spostamento corrispondenti ai vari registri.

Questi programmi ci sono serviti per spiegare il funzionamento di un PIA, che in realtà, in un caso come questo, non sarebbe stato necessario: un buffer tri-state indirizzabile era più che sufficiente e, tra l'altro, meno costoso.

13-2. Un interruttore a posizioni multiple (a rotazione, a selettore o a thumbweel)

Interfacciamo un interruttore a posizioni multiple ad un microprocessore MC68000. Il terminale corrispondente alla posizione dell'interruttore è collegato a massa, mentre gli altri sono alti (stato logico uno).

La Figura 13-3 mostra il circuito necessario ad interfacciare un interruttore ad 8 posizioni, che utilizza tutti gli otto bit di una porta del PIA. Ciò che deve fare il programma è stabilire la posizione dell'interruttore e controllare eventuali cambiamenti. Bisogna tener conto di due situazioni particolari:

1. L'interruttore occupa una posizione intermedia, così nessun terminale è a massa
2. L'interruttore non ha ancora raggiunto la sua posizione finale.

Nel primo caso, basta attendere finchè un bit della porta non è zero, cioè finchè un terminale dell'interruttore non è a massa. Nel secondo caso, controlleremo nuovamente l'interruttore dopo un

breve intervallo (di 1 o 2 secondi) ed accetteremo l'input solo quando rimane costante. Un ritardo così breve non inciderà sui tempi di risposta del sistema. Come alternativa, possiamo servirci anche di un altro interruttore (cioè, un interrutore di carico) per far sapere al processore quando leggere l'interruttore a selettore.

Sul circuito della Figura 13-3 realizzeremo due operazioni:

a. Controllare l'interruttore finchè non occupa una posizone definitiva. Quindi determinare la posizione e salvare il suo valore binario in una locazione di memoria.
b. Attendere che la posizione dell'interruttore cambi e, quindi, memorizzarla in una locazione di memoria.

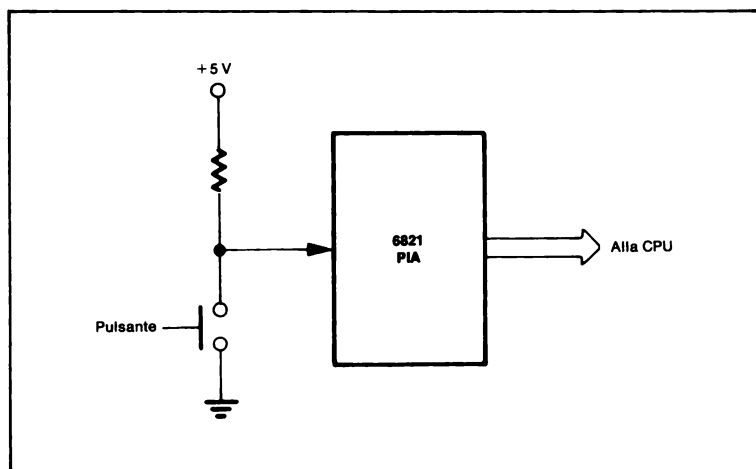



Figura 13-3. Un Interruttore a Posizioni Multiple.

Se l'interruttore si trova in una determinata posizione, il terminale corrispondente viene collegato a massa tramite la linea comune. La presenza di resistenze di pullup sulle linee di ingresso evita problemi dovuti al rumore.

Funzione 13-2a. Determinare la Posizione dell'Interruttore

Scopo: Il programma attende che l'interruttore si trovi in una determinata posizione e, quindi, memorizza il numero corrispondente nella variabile POSITION, alla locazione 6000.

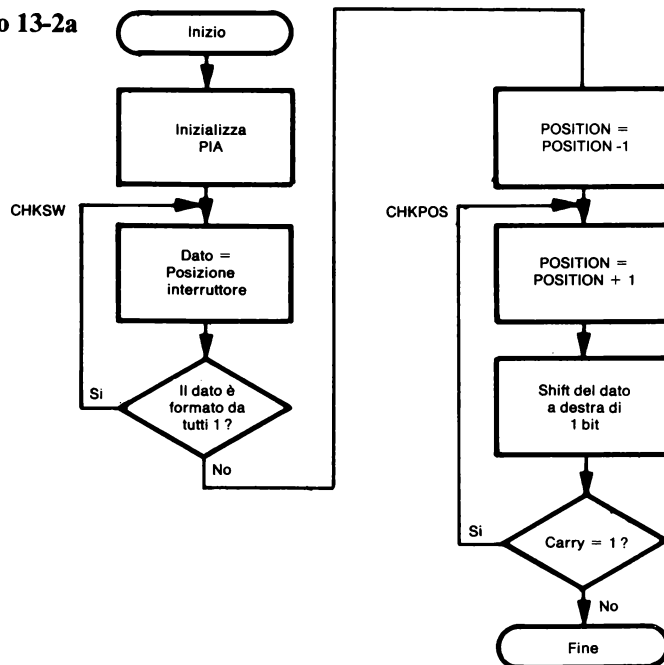
La Tabella 13-7 contiene gli input corrispondenti alle diverse posizioni dell'interruttore. Questo schema non è particolarmente efficiente, in quanto richiede otto bit per distinguere fra otto posizioni differenti.

Abbiamo cercato di aumentare l'efficacia del loop che identifica la posizione dell'interruttore. Dopo aver inizializzato la posizione con -1, la incrementiamo (con ADDQ #1) e shiftiamo l'input (con LSR). Cosa accadrebbe inizializzando con zero la posizione ed

effettuando lo shift ed il controllo prima di incrementarla? Il metodo che abbiamo adottato è molto comune e serve a ridurre il tempo di esecuzione del ciclo, perchè ci permette di trattare la prima iterazione allo stesso modo di quelle successive.

Programma 13-2a:

```
00004000:                PROGRAM  EQU     $4000
00006000:                DATA     EQU     $6000
                         ;
0003FF40:                PIA      EQU     $3FF40        INDIRIZZO DI BASE DEL PIA
                         ;
00000000:                PIADDA   EQU     $0            OFFSET REG. DIREZIONE DATI A
00000000:                PIADA    EQU     $0            OFFSET REG. DATI A
00000004:                PIACA    EQU     $4            OFFSET REG. DI CONTROLLO A
                         ;
00000000:                A_DATDIR EQU     $00           NEL LATO A SONO TUTTI INGRESSI
00000004:                A_CNTRL  EQU     $04           SELEZIONA REGISTRO DATI
                         ;
000000FF:                MASKNS   EQU     $FF           MASCHERA NESSUN INTERR. SELEZIONATO
                         ;
                                  ORG     DATA

00006000:                POSITION DS.B    1             POSIZIONE INTERRUTTORE
                         ;
                                  ORG     PROGRAM

00004000: 207C 0003
00004004: FF40           PGM13_2A MOVEA.L #PIA,A0       PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004               CLR.B   PIACA(A0)     INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                    MOVE.B  #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                    MOVE.B  #A_CNTRL,PIACA(A0)
                         ;
00004016: 1028 0000      CHKSW    MOVE.B  PIADA(A0),D0  LEGGI POSIZIONE INTERRUTTORE
0000401A: 0C00 00FF               CMP.B   #MASKNS,D0    E' UNA POSIZIONE DEFINITA?
0000401E: 67F6                    BEQ     CHKSW         SE NO, ATTENDI CHE LO DIVENTI
                         ;
00004020: 11FC 00FF
00004024: 6000                    MOVE.B  #-1,POSITION  ...ALTRIMENTI POSIZIONE INIZIALE
00004026: 5238 6000      CHKPOS   ADDQ.B  #1,POSITION   INCREMENTA POSIZIONE INTERRUTTORE
0000402A: E208                    LSR.B   #1,D0         IL BIT SUCCESSIVO E' A MASSA?
0000402C: 65F8                    BCS     CHKPOS        SE NO, CONTINUA LA RICERCA
                         ;
0000402E: 4E75                    RTS

                                  END     PGM13_2A
```

Diagramma di Flusso 13-2a

Per stabilire rapidamente se l'interruttore si trova in una posizione definita potremmo servirci di una sequenza di questo tipo:

```
CHKSW   ADDQ.B   #1,PIADA(A0)  L'INTERR. È IN UNA POSIZ. DEFI-
                               NITA?
        BEQ      CHKSW         ...NO, ATTENDI FINCHÈ NON C'È
```

Perchè questo metodo funziona? Il contenuto del registro dati del PIA cambia veramente? Sarebbe possibile usare il flag di Carry, anzichè il flag di Zero? Cercate di spiegare i motivi.

Tabella 13-7. Dati di Input rispetto alla Posizione dell'Interruttore

Posizione dell'Interruttore	Input	
	Binario	Esadecimale
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F

Non è uno schema efficiente, poichè richiede otto bit per distinguere fra otto posizioni diverse.

Questo esempio presuppone che i rimbalzi dell'interruttore siano eliminati dall'hardware. Quali modifiche si renderebbero necessarie per eliminare il fenomeno dei rimbalzi dall'interno del programma?

Con un codificatore TTL o MOS potremo ridurre il numero dei bit di input necessari. La Figura 13-4 mostra un circuito che impiega il codificatore TTL 8-3 74LS1483. Dato che questo dispositivo ha ingressi ed uscite attivi bassi, collegheremo le uscite dell'interruttore in odine inverso. Otterremo un output consistente nella rappresentazione a 3 bit della posizione dell'interruttore. Molti interruttori

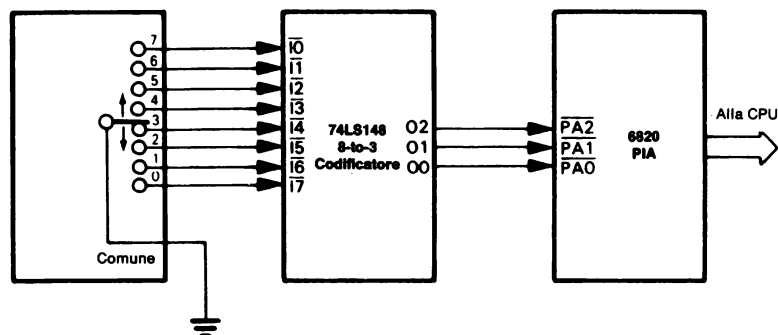



Figura 13-4. Un Interruttore a Posizioni Multiple con Codificatore.

contengono già dei codificatori, con output codificati di solito come cifre BCD (in logica negativa).

Il codificatore produce uscite basse attive, per cui, ad esempio, la posizione 5, corrispondente all'ingresso 2, fornisce un valore di output uguale a 2 in logica negativa (5 in logica positiva).

Supponiamo che l'interruttore o il PIA siano difettosi e facciano sì che l'input sia sempre ugale a FF_{16}. In che modo il programma potrebbe tener conto di questa eventualità?

Funzione 13-2b. Attendere il Cambiamento di Posizione dell'Interruttore

Scopo: Il programma attende che la posizione dell'interruttore cambi e mette la nuova posizione (decodificata) nella variabile POSITION, alla locazione di memoria 6000. Il programma attende che l'interruttore raggiunga la nuova posizione.

Diagramma di Flusso 13-2b

Programma 13-2b

```
00004000:                        PROGRAM   EQU     $4000
00006000:                        DATA      EQU     $6000
                                 ;
0003FF40:                        PIA       EQU     $3FF40          INDIRIZZO DI BASE DEL PIA
                                 ;
00000000:                        PIADDA    EQU     $0              OFFSET REG. DIREZIONE DATI A
00000000:                        PIADA     EQU     $0              OFFSET REG. DATI A
00000004:                        PIACA     EQU     $4              OFFSET REG. DI CONTROLLO A
                                 ;
00000000:                        A_DATDIR  EQU     $00             NEL LATO A SONO TUTTI INGRESSI
00000004:                        A_CNTRL   EQU     $04             SELEZIONA REGISTRO DATI
                                 ;
000000FF:                        MASKNS    EQU     $FF             MASCHERA NESSUN INTERR. SELEZIONATO
                                 ;
                                           ORG     DATA

00006000:                        POSITION  DS.B    1               POSIZIONE INTERRUTTORE
                                 ;
                                           ORG     PROGRAM

00004000: 207C 0003
00004004: FF40                   PGM13_2B  MOVEA.L #PIA,A0         PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004                        CLR.B   PIACA(A0)       INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                             MOVE.B  #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                             MOVE.B  #A_CNTRL,PIACA(A0)
                                 ;
00004016: 1028 0000              CHKSW1    MOVE.B  PIADA(A0),D0    LEGGI POSIZIONE INTERRUTTORE
0000401A: 0C00 00FF                        CMP.B   #MASKNS,D0      E' UNA POSIZIONE DEFINITA?
0000401E: 67F6                             BEQ     CHKSW1          SE NO, ATTENDI CHE LO DIVENTI
                                 ;
00004020: 1228 0000              CHKSW2    MOVE.B  PIADA(A0),D1    LEGGI NUOVO DATO INTERRUTTORE
00004024: 0C01 00FF                        CMP.B   #MASKNS,D1      L'INTERR. E' IN UNA POS. DEFINITA?
00004028: 67F6                             BEQ     CHKSW2          SE NO, ATTENDI CHE LA ASSUMA
                                 ;
0000402A: B200                             CMP.B   D0,D1           E' LA STESSA POSIZIONE DI PRIMA?
0000402C: 67F2                             BEQ     CHKSW2          SE SI', ATTENDI CHE CAMBI
                                 ;
0000402E: 11FC 00FF
00004032: 6000                             MOVE.B  #-1,POSITION    ...ALTRIMENTI POSIZIONE INIZIALE
00004034: 5238 6000              CHKPOS    ADDQ.B  #1,POSITION     INCREMENTA POSIZIONE INTERRUTTORE
00004038: E209                             LSR.B   #1,D1           IL BIT SUCCESSIVO E' A MASSA?
0000403A: 65F8                             BCS     CHKPOS          SE NO, CONTINUA LA RICERCA
                                 ;
0000403C: 4E75                             RTS
                                           END     PGM13_2B
```

13-3. Un led singolo

Interfacciamo un LED (Ligth Emitting Diode) ad un microprocessore MC68000 con due diversi programmi, uno per una gestione in logica positiva (un '1" lo accende) e l'altro in quella negativa (un '1' lo spegne).

La Figura 13-5 mostra il circuito necessario. Un LED si accende quando il suo anodo è positivo rispetto al catodo (Figura 13-5a). Questo risultato potremo ottenerlo sia collegando a massa il catodo mentre il computer fornisce una tensione positiva (uno logico) all'anodo (Figura 13-5b), oppure mettendo l'anodo a +5 volt e facendo in modo che il computer fornisca una tensione zero al catodo (Figura 13-5c). Il controllo del catodo rappresenta l'approccio più comune perché in questo modo le porte di I/O MOS e TTL forniscono prestazioni migliori. Un LED è più brillante quando funziona con correnti pulsanti di circa 10-50 mA, applicate alcune centinaia di volte al secondo. I LED hanno un tempo di accensione molto breve (dell'ordine di microsecondi) e ciò li rende particolarmente adatti al multiplexing (parecchi LED controllati da una sola porta). I circuiti a LED hanno bisogno di periferiche, di driver a transistor e di resistenze per limitare la corrente. Normalmente, i dispositivi MOS non possono pilotare direttamente dei LED e farli brillare abbastanza da renderli visibili.

Il PIA ha un latch d'uscita su ogni porta. Tuttavia, per l'output si preferisce utilizzare la porta B, in quanto dispone di una capacità di pilotaggio leggermente maggiore. Le uscite della porta B sono in grado di pilotare dei transistor Darlington (fornendo un minimo di 3,2mA a 1,5 V). I transistor Darlington sono dei transistor ad alto guadagno, capaci di commutare grandi quantità di corrente ad alta velocità; sono particolarmente utili per pilotare dei solenoidi, dei relè ed altri dispositivi del genere.

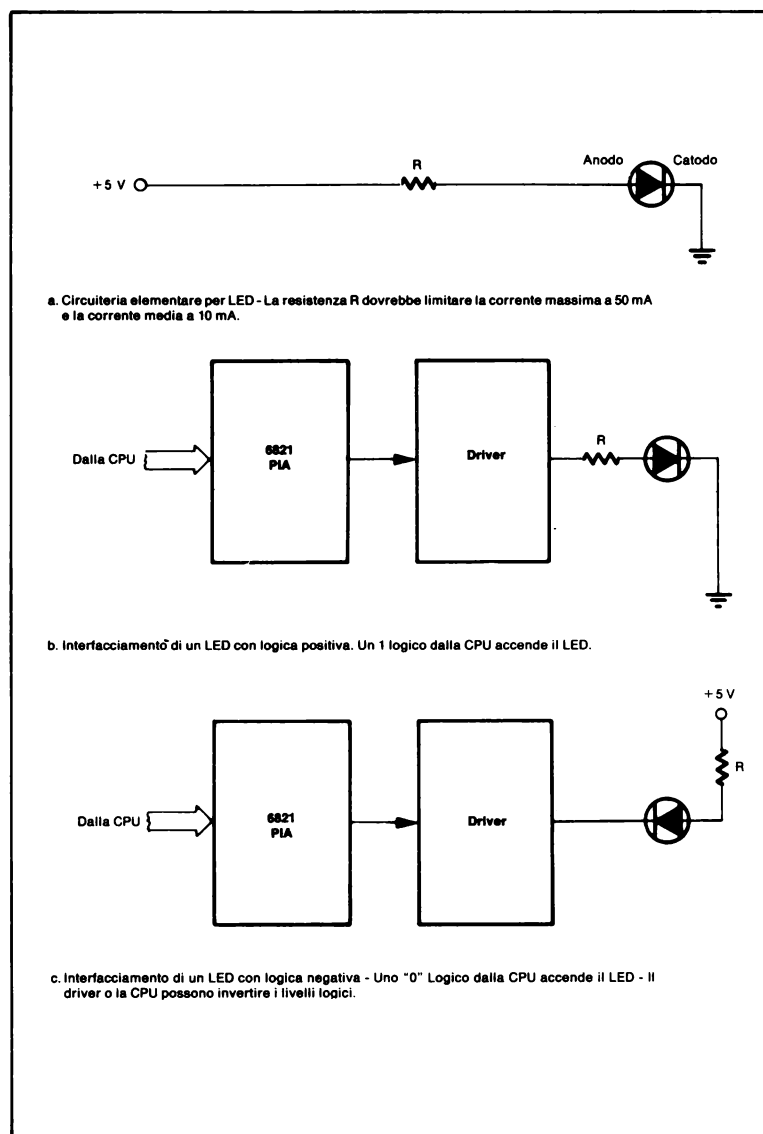



Figura 13-5. Interfacciamento di un LED.

Funzione 13-3. Accendere o Spengere un LED

Scopo: Il programma accende o spegne un unico LED. Se gli inviamo un '1' Logico, un LED si accenderà se opera in logica positiva mentre si spegnerà se opera in logica negativa.

Programma 13-3:

```
00004000:                PROGRAM  EQU      $4000
00006000:                DATA     EQU      $6000
                         ;
0003FF40:                PIA      EQU      $3FF40            INDIRIZZO DI BASE DEL PIA
                         ;
00000002:                PIADDB   EQU      $2                OFFSET REG. DIREZIONE DATI B
00000002:                PIADB    EQU      $2                OFFSET REG. DATI B
00000006:                PIACB    EQU      $6                OFFSET REG. DI CONTROLLO B
                         ;
00000080:                B_DATDIR EQU      $80               IL BIT 7 DEL LATO B E' UN'USCITA
00000004:                B_CNTRL  EQU      $04               SELEZIONA REGISTRO DATI
                         ;
00000007:                LEDBIT   EQU      7                 POSIZIONE DEL BIT DEL LED NEL PIA
                         ;
                                  ORG      PROGRAM

00004000: 207C 0003
00004004: FF40           PGM13_3  MOVEA.L  #PIA,A0           PRENDI IND. DI BASE DEL PIA
00004006: 4228 0006               CLR.B    PIACB(A0)         INIZIALIZZA IL LATO B
0000400A: 117C 0080
0000400E: 0002                    MOVE.B   #B_DATDIR,PIADDB(A0)
00004010: 117C 0004
00004014: 0006                    MOVE.B   #B_CNTRL,PIACB(A0)
                         ;
00004016: 08E8 0007
0000401A: 0002                    BSET     #LEDBIT,PIADB(A0)   PREPARA LED
                         ;
0000401C: 4E75                    RTS
                         ;
                         ;    *   (AGGIORNA DATO)
                         ;
0000401E: 08E8 0007
00004022: 0002                    BSET     #LEDBIT,PIADB(A0) PONI A 1 IL BIT DI OUTPUT DEL LED
                         ;
00004024: 4E75                    RTS

                                  END      PGM13_3
```

Utilizziamo il lato B del PIA perchè ci dà la possibilità di disporre di un buffer e, se necessario, la CPU potrà rileggere un dato senza alcuna difficoltà.

13-4. Display led a sette segmenti

Interfacciamo un display LED a sette segmenti ad un microprocessore MC68000. Può trattarsi di un display ad anodo comune (logica negativa) oppure a catodo comune (logica positiva).

La Figura 13-6 mostra il circuito necessario. Ad ogni segmento possono corrispondere uno, due o più LED collegati allo stesso modo. Sono possibili due tipi di collegamento. Uno consiste nel **collegare tutti i catodi insieme a massa** (vedi Figura 13-7a), realizzando **un "display a catodo comune", in cui un uno logico all'anodo provocherà l'accensione di un segmento.** L'altro consiste nel **collegare tutti gli anodi ad una tensione di alimentazione positiva** (vedi Figura 13-7b); questo è **un "display a anodo comune" ed uno zero logico al catodo accende un segmento.** Riassumendo, un display a catodo

comune impiega una logica positiva, quello ad anodo comune una logica negativa. Entrambi richiedono driver e resistenze appropriati.

La linea comune proveniente dal display è collegata a massa oppure a +5 volt. Ad ogni segmento del display viene associata una lettera, in modo del tutto arbitrario:

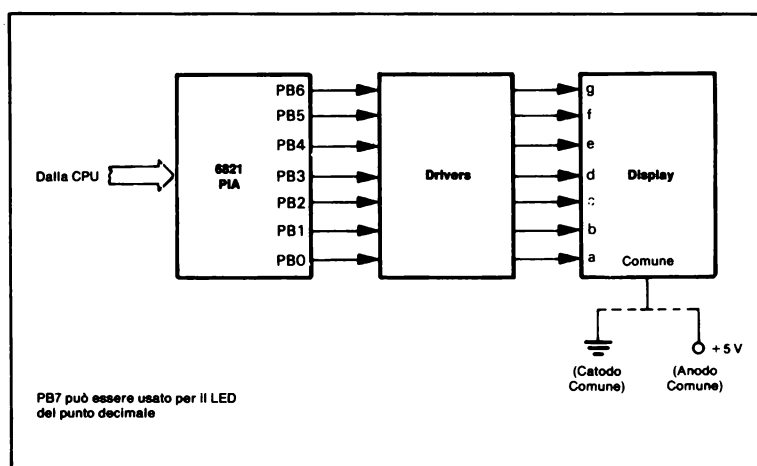
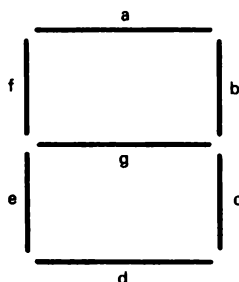



Figura 13-6. Interfacciamento di un Display a Sette Segmenti.

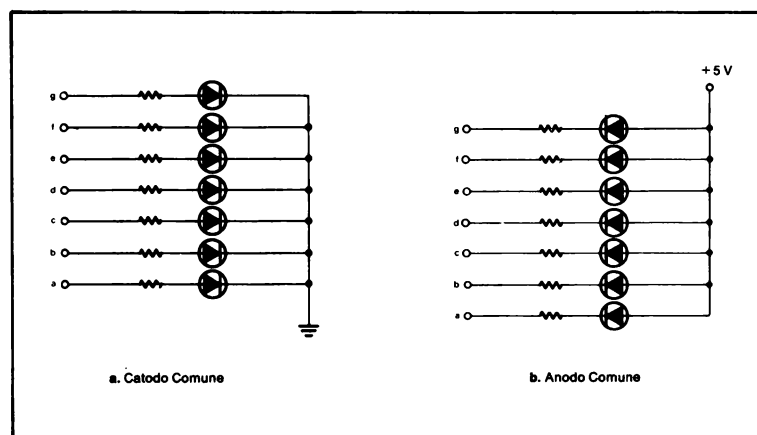



Figura 13-7. Organizzazione di un Display a Sette Segmenti.

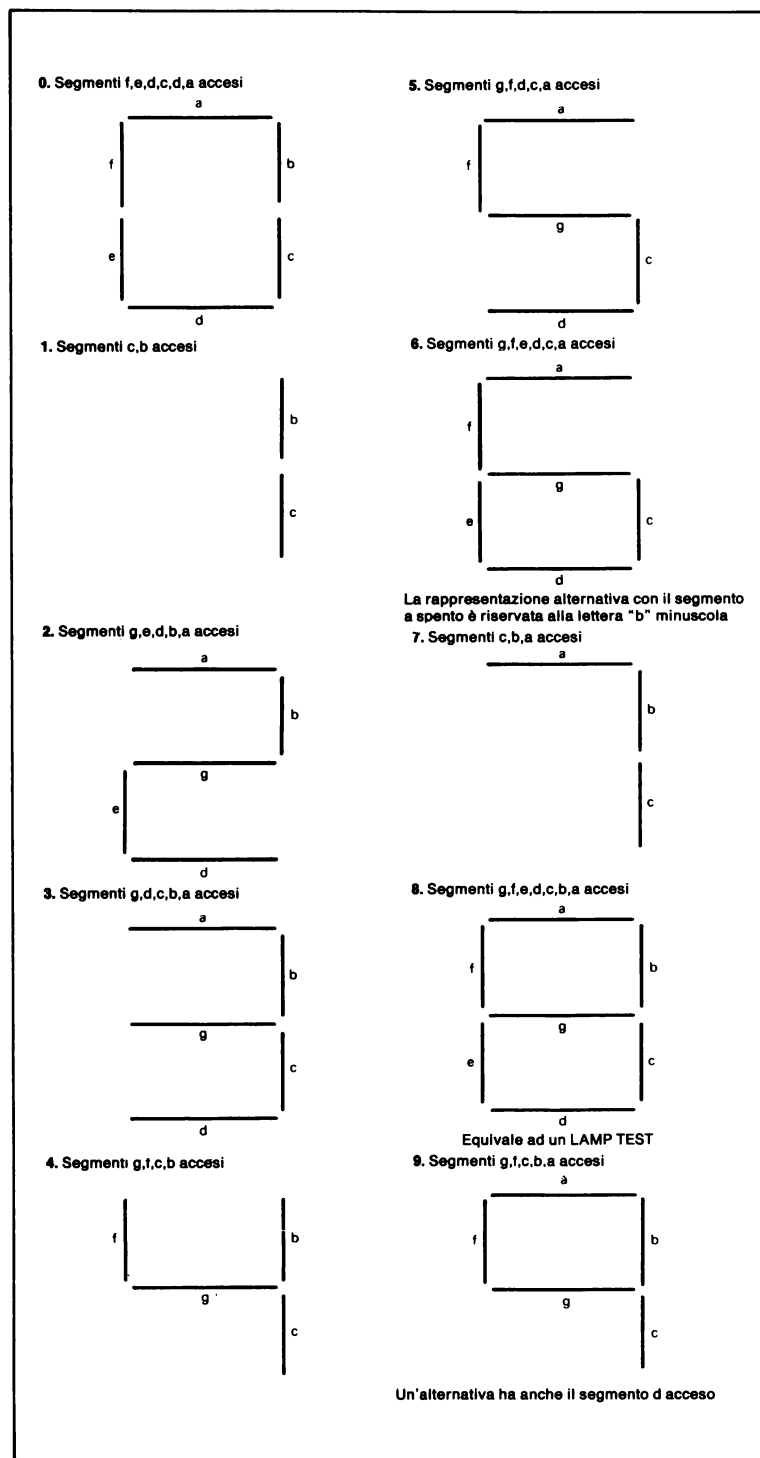



Figura 13-8. Rappresentazione a sette segmenti delle Cifre Decimali.

Display come questo sono molto usati, perchè contengono il numero minimo di segmenti controllabili separatamente, in grado di rappresentare in modo riconoscibile tutte le cifre decimali (cfr. Figura 13-8 e Tabella 13-8), oltre ad alcune lettere e ad altri caratteri (cfr. Tabella 13-9). Rappresentazioni migliori richiederebbero un numero notevolmente maggiore di segmenti e di circuiti.[4] **Data la diffusione dei display a sette segmenti, anche i relativi decodificatori/driver sono diventati disponibili a basso prezzo.** I dispositivi più adottati sono il driver ad anodo comune 7447 ed il driver a catodo comune 7448[5], che dispongono di ingressi Lamp Test (che accendono tutti i segmenti) ed ingressi e uscite di spegnimento (per la soppressione degli zeri non significativi, iniziali e finali).

Tabella 13-8. Rappresentazione a sette segmenti dei Numeri Decimali

Numero	Rappresentazione Esadecimale	
	Catodo Comune	Anodo Comune
0	3F	40
1	06	79
2	5B	24
3	4F	30
4	66	19
5	6D	12
6	7D	02
7	07	78
8	7F	00
9	67	18
Il Bit 7 è sempre zero e gli altri sono g,f,e,d,c,b ed a in ordine decrescente di significato		

Tabella 13-9. Rappresentazione a sette segmenti di Lettere e Simboli

Lettere maiuscole			Lettere minuscole e Caratteri Speciali		
Lettera	Rappresentazione Esadecimale		Carattere	Rappresentazione Esadecimale	
	Catodo Comune	Anodo Comune		Catodo Comune	Anodo Comune
A	77	08	b	7C	03
C	39	46	c	58	27
E	79	06	d	5E	21
F	71	0E	h	74	0B
H	76	09	n	54	2B
I	06	79	o	5C	23
J	1E	61	r	50	2F
L	38	47	u	1C	63
O	3F	40	-	40	3F
P	73	0C	?	53	2C
U	3E	41			
Y	66	19			

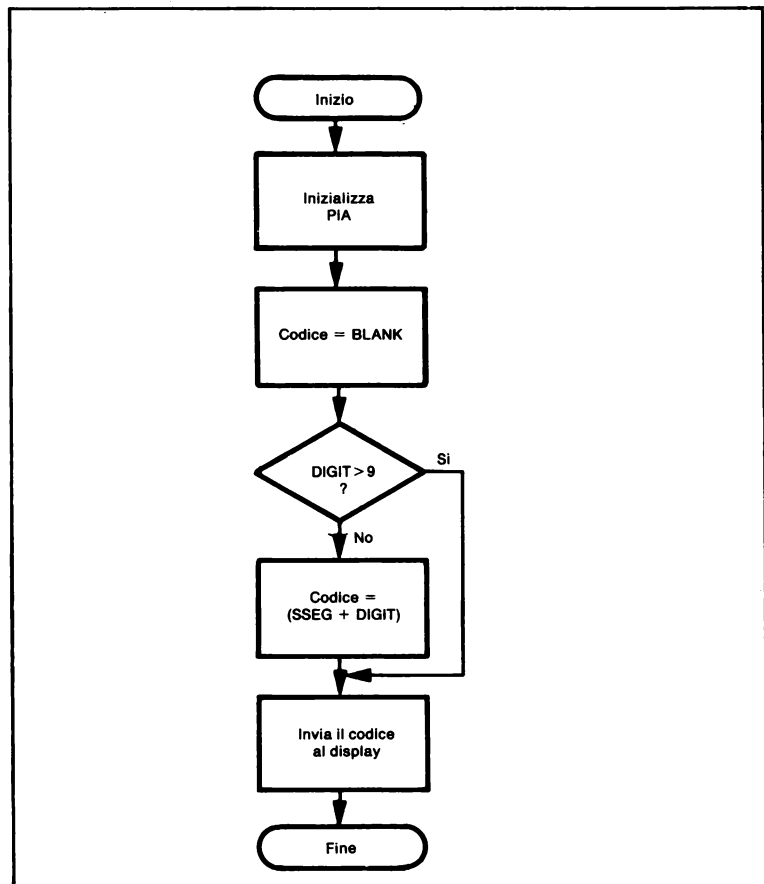
Funzione 13-4a. Visualizzare una Cifra Decimale

Scopo: Visualizzare il contenuto della variabile DIGIT, alla locazione di memoria 6000, su un display a sette segmenti, solo se si tratta di una cifra decimale. Altrimenti cancellare il display.

Prblemi Campione:

a. DIGIT = (6000) = 05
Il risultato è un 5 sul display

b. DIGIT = (6000) = 66
Il risultato è un display vuoto

Diagramma di Flusso 13-4a

Programma 13-4a

```
00004000:                      PROGRAM  EQU      $4000
00006000:                      DATA     EQU      $6000
                               ;
0003FF40:                      PIA      EQU      $3FF40           INDIRIZZO DI BASE DEL PIA
                               ;
00000002:                      PIADDB   EQU      $2               OFFSET REG. DIREZIONE DATI B
00000002:                      PIADB    EQU      $2               OFFSET REG. DATI B
00000006:                      PIACB    EQU      $6               OFFSET REG. DI CONTROLLO B
                               ;
000000FF:                      B_DATDIR EQU      $FF              TUTTE USCITE SUL LATO B
00000004:                      B_CNTRL  EQU      $04              SELEZIONA REGISTRO DATI
                               ;
                                        ORG      DATA

00006000:                      DIGIT    DS.B     1                DATO DA VISUALIZZARE
                               ;
00006001:                      SSEG     DC.B     $3F,$06,$5B,$4F  CODICE PER CIFRE A SETTE
00006005:                               DC.B     $66,$6D,$7D,$07  SEGMENTI SU UN COMUNE
00006009:                               DC.B     $7F,$67          DISPLAY CRT
00000000:                      BLANK    EQU      $00              SPAZIO  SU UN COMUNE CRT
                                        ORG      PROGRAM

00004000: 207C 0003
00004004: FF40                 PGM13_4A MOVEA.L  #PIA,A0          PRENDI IND. DI BASE DEL PIA
00004006: 4228 0006                     CLR.B    PIACB(A0)        INIZIALIZZA IL LATO B
0000400A: 117C 00FF
0000400E: 0002                          MOVE.B   #B_DATDIR,PIADDB(A0)
00004010: 117C 0004
00004014: 0006                          MOVE.B   #B_CNTRL,PIACB(A0)
00004016: 103C 0000                     MOVE.B   #BLANK,D0        PRENDI CODICE SPAZIO
0000401A: 4241                          CLR.W    D1               AZZERA INDICE (D1.W)
0000401C: 1238 6000                     MOVE.B   DIGIT,D1         PRENDI IL DATO
00004020: 0C01 0009                     CMPI.B   #9,D1            E' UNA CIFRA DECIMALE(9 O INF.)?
00004024: 620A                          BHI.S    DSPLY            SE NO, VISUALIZZA SPAZIO
                               ;
00004026: 227C 0000
0000402A: 6001                          MOVEA.L  #SSEG,A1         INDIRIZZO TABELLA SEGMENTI
0000402C: 1031 1000                     MOVE.B   0(A1,D1.W),D0    CONVERTI IL DATO PER VISUALIZZARLO
00004030: 1140 0002            DSPLY    MOVE.B   D0,PIADB(A0)     INVIA IL CODICE AL DISPLAY
                               ;
00004034: 4E75                          RTS

                                        END      PGM13_4A
```

Bisogna notare due aspetti in questo programma. Innanzitutto, volendo adottare un display ad anodo comune la sola cosa da cambiare è il contenuto della tabella SSEG ed il valore del simbolo BLANK. In secondo luogo non va dimenticato che BLANK è un simbolo e non è contenuto in memoria come la tabella SSEG. Una soluzione alternativa consiste nel mettere anche BLANK in memoria, come ultimo elemento della tabella SSEG. In questo caso, andranno sostituiti alcuni dati con 10 e, di conseguenza, le istruzioni successive a MOVE.B #B–CNTRL,PIACB(A0) diventeranno:

```
00006000:                      SSEG     EQU      DATA

00004016: 1238 6000                     MOVE.B   DATA,D1          PRENDI IL DATO
0000401A: 0C01 0009                     CMPI.B   #9,D1            E' UNA CIFRA DECIMALE (9 O INF.)
0000401E: 6300 0006                     BLS      CNVRT            ...SI', CONVERTI
                               ;
00004022: 123C 000A                     MOVE.B   #10,D1           ..ALTRIMENTI INDICE CODICE SPAZIO
00004026: 207C 0000
0000402A: 6000                 CNVRT    MOVEA.L  #SSEG,A0         CONVERTI DATO A COD. SETTE SEG.
0000402C: 1170 1000
00004030: 0002                          MOVE.B   0(A0,D1),PIADB(A0)  INVIA CODICE AL DISPLAY
```

La Figura 13-9 mostra come multiplexare dei display (cioè, come pilotare parecchi display dalla stessa porta).[6] Un breve impulso sulla linea di controllo CB2 sincronizza automaticamente il contatore decimale dopo ogni operazione di output, indirizzando, così, il dato al display successivo. Il RESET inizializza il contatore decimale con

9, in modo che la prima operazione di output azzeri il contatore ed indirizzi il dato al primo display (in realtà, lo zero).

Il programma che segue impiega una routine di ritardo della durata di 1 ms. (descritta nel Capitolo 12) per inviare un impulso da 1 ms. a tutti i dieci display a catodo comune. Appariranno visualizzate contemporaneamente dieci cifre, come accade nei calcolatori tascabili, negli orologi e nei registratori di cassa.

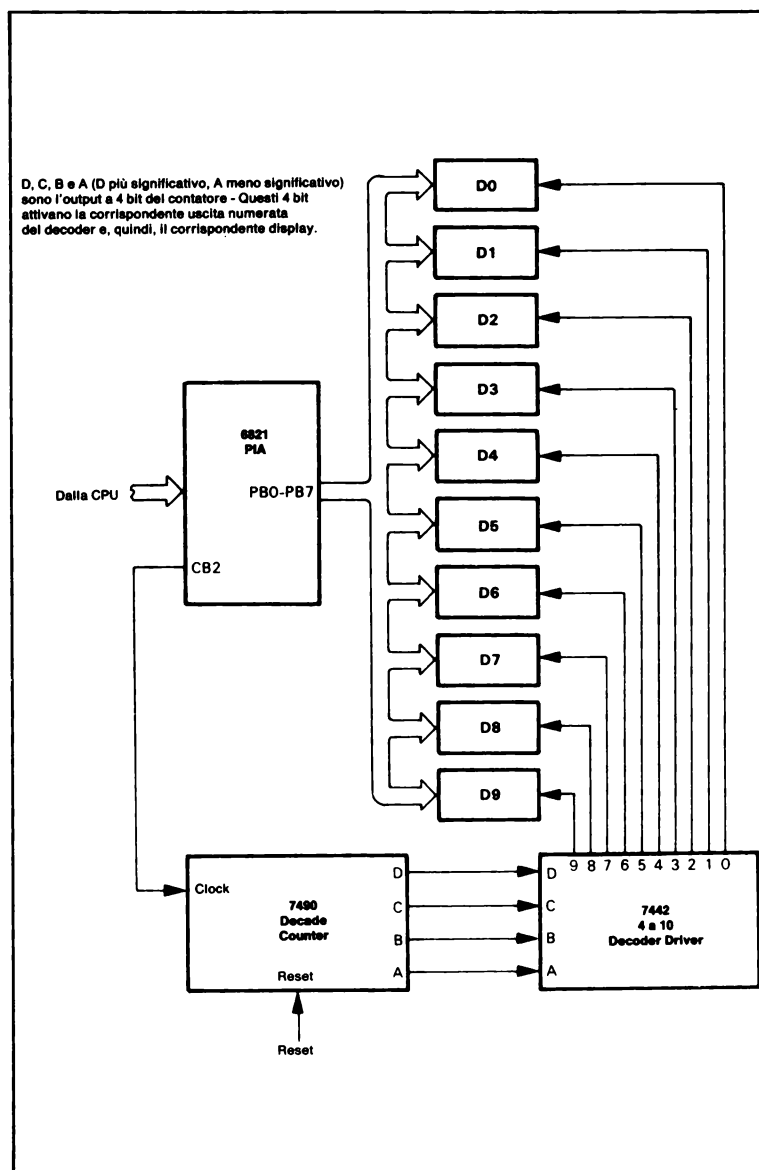



Figura 13-9. Display a Sette Segmenti Multiplexato.

Programma 13-4b

```
00004000:                   PROGRAM  EQU      $4000
00006000:                   DATA     EQU      $6000
00004100:                   DELAY    EQU      $4100              INDIRIZZO ROUTINE DI RITARDO
                            ;
0003FF40:                   PIA      EQU      $3FF40             INDIRIZZO DI BASE DEL PIA
                            ;
00000002:                   PIADDB   EQU      $2                 OFFSET REG. DIREZIONE DATI B
00000002:                   PIADB    EQU      $2                 OFFSET REG. DATI B
00000006:                   PIACB    EQU      $6                 OFFSET REG. DI CONTROLLO B
                            ;
000000FF:                   B_DATDIR EQU      $FF                TUTTE USCITE SUL LATO B
0000002C:                   B_CNTRL  EQU      $2C                SELEZIONA REGISTRO DATI, STROBE DI
                                                                 OUTPUT SU CB2 SCRIVENDO NEL REG. B
                            ;
00000001:                   DELAYTIM EQU      1                  TEMPO DI VISUALIZZAZIONE IN MS.
0000000A:                   NUMDIGIT EQU      10                 NUM. DI CIFRE DA VISUALIZZARE
                            ;
                                     ORG      DATA

00006000:                   DIGIT    DS.B     NUMDIGIT           DATO DA VISUALIZZARE
                            ;
                                     ORG      PROGRAM

00004000: 207C 0003
00004004: FF40              PGM13_4B MOVEA.L  #PIA,A0            PRENDI IND. DI BASE DEL PIA
00004006: 4228 0006                  CLR.B    PIACB(A0)          INIZIALIZZA IL LATO B
0000400A: 117C 00FF
0000400E: 0002                       MOVE.B   #B_DATDIR,PIADDB(A0)
00004010: 117C 002C
00004014: 0006                       MOVE.B   #B_CNTRL,PIACB(A0)
                            ;
00004016: 227C 0000
0000401A: 6000              SCAN     MOVEA.L  #DIGIT,A1          PUNTATORE INIZIO DATI
0000401C: 7209                       MOVEQ    #NUMDIGIT-1,D1     CONT. LOOP CORRETTO PER DBRA
                            ;
0000401E: 1159 0002         DISPLAY  MOVE.B   (A1)+,PIADB(A0)    SPOSTA IL DATO SUL DISPLAY
00004022: 7001                       MOVEQ    #DELAYTIM,D0       ATTENDI
00004024: 4EB8 4100                  JSR      DELAY
00004028: 51C9 FFF4                  DBRA     D1,DISPLAY         CONTINUA CON LE ALTRE CIFRE
                            ;
0000402C: 60E8                       BRA      SCAN               RIPETI SCAN

                                     END      PGM13_4B
```

Funzione 13-4b. Visualizzare Dieci Cifre Decimali

Scopo: Visualizzare stabilmente il contenuto del vettore DIGIT, che inizia alla locazione di memoria 6000, su dieci display a sette segmenti, multiplexati con un contatore ed un decodificatore. La cifra più significativa (quella più a sinistra) si trova nella locazione di memoria 6000.

Problema Campione:

```
DIGIT = (6000) = 66
        (6001) = 3F
        (6002) = 7F
        (6003) = 7F
        (6004) = 06
        (6005) = 5B
        (6006) = 07
        (6007) = 4F
        (6008) = 6D
        (6009) = 7D
```

Il numero sul display è 4088127356.

Nel registro di controllo Il bit 5 = 1 perchè CB2 deve essere un'uscita, il bit 4 = 0 perchè si deve trattare di impulso ed il bit 3 = 1 perchè l'impulso deve avere una durata pari ad un ciclo di clock. Quali cambiamenti sono necessari per visualizzare un numero diverso di cifre, ad esempio 8 cifre?

DISPOSITIVI DI I/O PIÙ COMPLESSI

I dispositivi di I/O più complessi si differenziano da delle semplici tastiere, interruttori e display per i seguenti motivi:

1. **Trasferiscono i dati a velocità più elevate.**
2. **Possono disporre di clock e temporizzatori interni.**
3. **Producono informazioni di stato e richiedono informazioni di controllo, oltre ad effettuare un normale trasferimento di dati.**

Proprio per le le elevate velocità di trasferimento, questi dispositivi non possono essere gestiti in modo casuale, altrimenti il sistema può perdere dei dati in entrata o produrre dei dati in uscita completamente errati. È necessario, quindi, tener conto delle particolari esigenze di queste periferiche, molto più di quanto non avvenga con apparecchiature più semplici. Una soluzione largamente impiegata è quella che prevede l'uso degli interrupt, come vedremo meglio nel Capitolo 15.

SINCRONIZZAZIONE

Periferiche come le tastiere, le telescriventi, i registratori a cassette ed i floppy disk dispongono di un proprio temporizzatore interno. Questi dispositivi generano dei flussi di dati, separati da determinati intervalli di tempo. **L'elaboratore deve sincronizzare la prima operazione di input, o di output, con il clock della periferica e, quindi, fornire l'intervallo richiesto fra due operazioni successive con una normale routine di ritardo, come quella che vi abbiamo descritto in precedenza. La sincronizzazione richiede una o più delle seguenti procedure:**

Procedure di sincronizzazione

1. **Rilevare la presenza di una transizione su una linea di clock o di strobe proveniente dalla periferica, ai fini della temporizzazione.** Il metodo più semplice è quello di collegare lo strobe ad una linea di controllo del PIA ed attendere finchè il corrispondente bit del registro di controllo non diventa 1.
2. **Determinare il centro dell'intervallo di tempo durante il quale il dato è stabile.** È preferibile leggere il valore di un dato al centro di un impulso piuttosto che in corrispondenza dei fronti, quando il dato sta cambiando. Per trovare il centro basta calcolare la

metà del tempo di trasmissione richiesto da un bit. Campionando un dato al centro, si riduce anche l'incidenza che piccoli errori di temporizzazione possono avere sull'accuratezza della ricezione.

3. **Riconoscere uno speciale codice d'inizio.** Questo è facile se il codice è di un solo bit o se disponiamo di segnali di temporizzazione. La procedura diventa più complessa se il codice è lungo e può iniziare in un momento qualsiasi. Saranno necessari anche degli shift per stabilire quando il trasmettitore inizia l'invio di un bit, di un carattere o di un messaggio. È la cosiddetta ricerca di un "framing" (lett. inquadratura) corretto.
4. **Campionare ripetutamente un dato.** Questo riduce la probabilità di ricevere dati errati su linee disturbate. Può essere impiegata la logica maggioritaria (al meglio di 3 su 5 o di 5 su 8) per stabilire il reale valore di un dato.

Naturalmente, la ricezione è molto più difficile della trasmissione, in quanto è controllata dalla periferica ed il computer si limita ad interpretare le informazioni di temporizzazione che gli sono inviate. Nel caso della trasmissione, è, invece, l'elaboratore a fornire la temporizzazione e il formato necessari ad una particolare periferica.

INFORMAZIONI DI CONTROLLO E DI STATO

Informazioni di controllo

Informazioni di stato

Le periferiche possono fornire o richiedere altre informazioni oltre ai dati ed agli impulsi di temporizzazione. Quando è l'elaboratore che invia queste informazioni, parleremo di "informazioni di controllo", che hanno lo scopo di selezionare un modo operativo, di iniziare o arrestare processi di elaborazione, sincronizzare dei registri, abilitare dei buffer, definire formati o protocolli, fornire delle visualizzazioni per l'operatore, contare il numero delle operazioni o identificare il tipo e la priorità di un'operazione. **L'altro tipo di informazioni inviate da una periferica vengono dette "informazioni di stato".** Servono ad indicare il modo operativo, la disponibilità a ricevere un nuovo dato, la presenza di eventuali condizioni d'errore, il protocollo utilizzato, ecc.

Il computer considera le informazioni di controllo e di stato al pari degli altri dati. Queste informazioni cambiano raramente, anche aumentando notevolmente la velocità di trasferimento dei dati, e sono costituite da singoli bit, da cifre oppure da byte singoli o multipli.

Come interpretare le Informazioni di Stato

Combinando le informazioni di stato e di controllo all'interno di singoli byte si riduce il numero totale di porte di I/O necessarie per una periferica. Tuttavia, questo significa dover interpretare separa-

tamente i singoli bit di stato in ingresso e definire singolarmente i bit di controllo in uscita. **Questa è la procedura per isolare i bit di stato:**

Procedura di isolamento dei bit di stato

Fase 1. Leggere i dati di stato dalla periferica.
Fase 2. AND logico con una maschera, che avrà degli uno in corrispondenza delle posizioni da esaminare e degli zeri nelle altre.
Fase 3. Shiftare i bit rimasti a 1 verso le posizioni meno significative.

La Fase 3 non è necessaria se si tratta di un unico bit, in quanto il flag di Zero conterrà il complemento di quel bit. (Provate!). Possiamo sostituire la Fase 2 con un'istruzione di shift o di caricamento se il campo è costituito da un solo bit che, all'interno del byte, occupa la posizione meno significativa, più significativa oppure quella accanto alla più significativa, cioè, rispettivamente le posizioni 0, 7 e 6 che sono, poi, quelle utilizzate di solito per le informazioni di stato. Provate a scrivere le istruzioni necessarie per l'MC68000. In casi come questo si rivela particolarmente utile l'istruzione BTST, che esegue un test logico del bit indicato senza modificarlo; il flag di stato Zero varia nel modo seguente:

flag di Zero = 1 se il bit è zero
flag di Zero = 0 se il bit è uno

Il Formato delle Informazioni di Controllo

Procedura di formazione e invio dei bit di controllo

Questa è la normale procedura per porre ad uno o azzerare i bit di controllo:

Fase 1. Leggere le precedenti informazioni di controllo.
Fase 2. AND logico con una maschera per azzerare determinati bit (la maschera avrà degli zeri nelle posizioni dei bit da azzerare, degli uno nelle altre).
Fase 3. OR logico con una maschera per mettere ad uno dei singoli bit (la maschera avrà degli uno nelle posizoni dei bit da porre ad uno, degli zeri nelle altre).
Fase 4. Inviare le informazioni di controllo alla periferica.

Anche in questo caso la procedura diventa più semplice se il campo è di un solo bit ed occupa una delle posizioni più esterne di un byte.

Ecco alcuni esempi per separare e combinare dei bit di stato tra loro:

1. Un campo di 3 bit, che occupano le posizioni 2, 3 e 4 di un registro dati del PIA, contiene un fattore scalare. Mettere questo valore nel registro dati D0.

```
                        *
                        * LEGGE LE INFORMAZIONI DI STATO DALLA PORTA DI INPUT
                        *
004164 10280000                MOVE.B  PIADA(A0),D0    LEGGE LE INFORMAZIONI DI STATO
                        *
                        * ESCLUDE I BIT SCARTATI E SPOSTA I RISULTATI
                        *
004168 0200001C                ANDI.B  =S1C,D0         SALVA IL FATTORE DI SCALA
00416C E408                    LSR.B   =2,D0           TRASFERISCE DUE VOLTE PER
                                                       NORMALIZZARE
```

2. Il registro dati D0 contiene un campo di 2 bit che devono essere messi nelle posizioni 3 e 4 del registro dati di un PIA.

```
                        *
                        * SPOSTA I DATI NELLA POSIZIONE DEL CAMPO
                        *
00416E E708                    LSL.B   =3,D0           TRASFERISCE I DATI ALLE
                                                       POSIZIONI 3 E 4
004170 02000018                ANDI.B  =S18,D0         AZZERA GLI ALTRI BIT
                        *
                        * UNISCE LA NUOVA POSIZIONE DEL CAMPO CON GLI ALTRI DATI
                        *
004174 022800E70000            ANDI.B  =SE7,PIADA(A0)  AZZERA IL VECCHIO VALORE DEL
                                                       CAMPO
00417A 81280000                OR.B    D0,PIADA(A0)    INSERISCE IL NUOVO VALORE DEL
                                                       CAMPO
```

Commentare i Trasferimenti di Informazioni di Stato e di Controllo

La documentazione è un aspetto estremamente importante anche nel caso delle informazioni di stato e di controllo, il cui significato non sempre riusciamo ad interpretare con facilità. È necessario, perciò, indicare, con estrema chiarezza, le finalità delle operazioni di input/output con dei commenti, del tipo, "CONTROLLA SE IL LETTORE È ACCESO", "SCEGLI LA PARITÀ DI TIPO PARI" oppure "ATTIVA IL CONTATORE DI VELOCITÀ". Altrimenti, le varie istruzioni logiche o di shift risulterebbero praticamente incomprensibili e diventerebbe molto difficile correggerle.

13-5. Una tastiera non codificata

Il processore rileva l'avvenuta pressione di un tasto su una tastiera 3 x 3 non codificata e mette il numero corrispondente al tasto premuto nel registro dati D0.

Le tastiere non sono altro che un insieme di interruttori (cfr. la Figura 13-10). È facilissimo gestire un piccolo numero di tasti se ognuno di essi è collegato ad un bit di una porta di input, per cui interfacciare una tastiera equivale ad interfacciare una serie di interruttori.

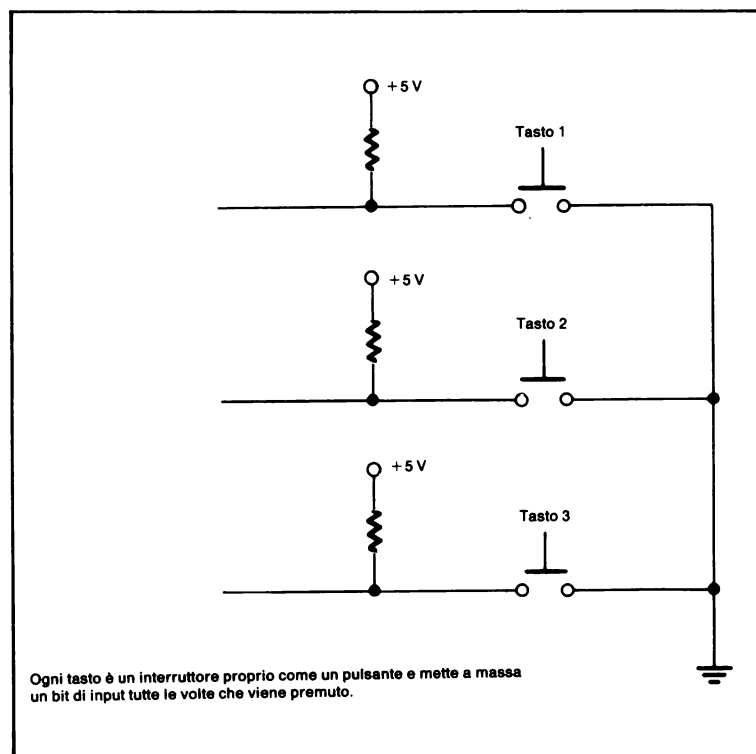



Figura 13-10. Una Piccola Tastiera.

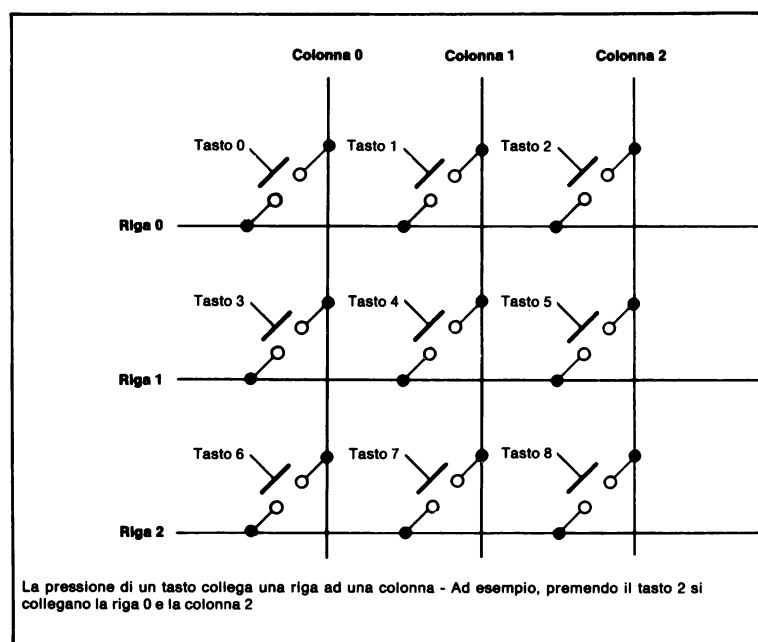



Figura 13-11. Una Matrice per Tastiera.

Tastiera a Matrice

Le tastiere con un numero di tasti superiore ad otto richiedono più di una porta di input e, quindi, per gestirle sono necessarie operazioni multibyte. La cosa diventa particolarmente complessa se non esiste un rapporto logico tra i vari tasti, come accade con la tastiera di una calcolatrice o di di un terminale, dove l'utente ne preme uno alla volta, in modo del tutto casuale. **Possiamo ridurre il numero delle linee d'ingresso necessarie collegando i tasti in forma di matrice, com'è illustrato nella Figura 13-11. In questo modo ogni tasto rappresenta un collegamento potenziale fra una riga ed una colonna.** La matrice di una tastiera richiede n + m linee esterne, dove n è il numero delle righe ed m il numero delle colonne, mentre, collegando separatamente ogni tasto, ce ne vorrebbero n x m. La Tabella 13-10 riporta il numero di tasti necessario nelle configurazioni più diffuse.

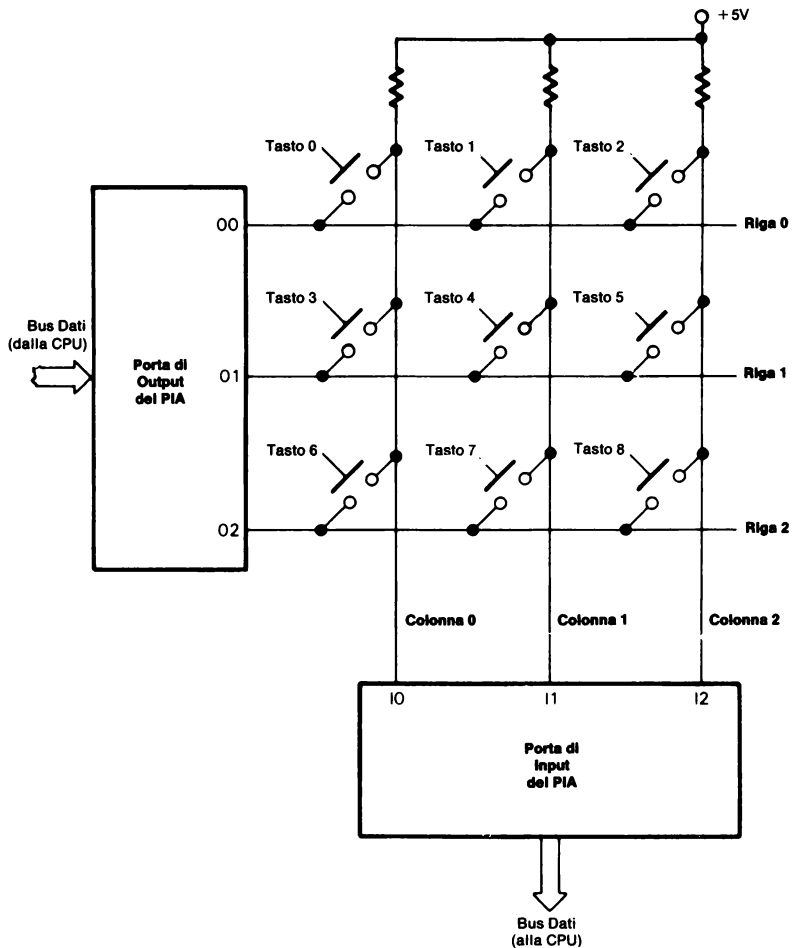



Figura 13-12. Disposizione dell'I/O per la Lettura di una Tastiera.

Tabella 13-10. Confronto fra Collegamenti indipendenti e Collegamenti a Matrice per una Tastiera.

Dimensione della Tastiera	Numero delle linee con Connessioni Indipendenti	Numero delle linee con Connessioni a Matrice
3 × 3	9	6
4 × 4	16	8
4 × 6	24	10
5 × 5	25	10
6 × 6	36	12
6 × 8	48	14
8 × 8	64	16

Esplorazione della Tastiera

Un programma riesce a stabilire quale tasto è stato premuto servendosi delle linee esterne della matrice. La procedura seguita è la cosiddetta "esplorazione della tastiera" (keyboard scan). Dopo aver collegato a massa la Riga 0, esaminiamo le linee corrispondenti alle colonne. Se una delle linee è a massa, significa che un tasto è stato premuto in quella riga, causando una connessione riga-colonna. Per individuare il tasto premuto, basta vedere quale colonna è stata messa a massa; cioè, quale bit della porta di input vale zero. Se nessuna colonna è a massa, passiamo alla Riga 1 e ripetiamo l'esplorazione. **Possiamo controllare se è stato premuto un tasto qualsiasi, collegando a massa tutte le righe contemporaneamente ed esaminando tutte le colonne.**

L'esplorazione di una tastiera richiede che le righe siano collegate ad una porta di output e le colonne ad una porta di input, come indicato dalla Figura 13-12. La CPU è in grado di mettere a massa una riga, mettendo uno zero nel corrispondente bit della porta di output e degli uno negli altri bit.

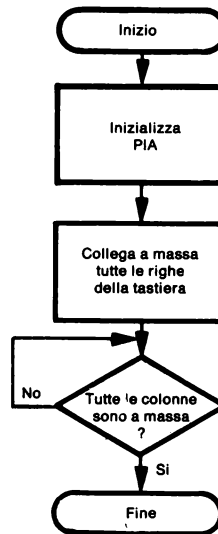
La CPU determina lo stato di una colonna, esaminando il corrispondente bit della porta di input.

Funzione 13-5a. Attesa della Chiusura di un Tasto

Scopo: Attendere che venga premuto un tasto.

La proceudra è la seguente:

1. Collegare a massa le righe azzerando tutti i bit di output.
2. Prelevare gli input delle colonne leggendo la porta di input.
3. Ritornare alla Fase 1 se tutti gli input delle colonne sono uno.

Diagramma di Flusso 13-5a

Programma 13-5a

```
00004000:                  PROGRAM  EQU      $4000
00006000:                  DATA     EQU      $6000
                           ;
0003FF40:                  PIA      EQU      $3FF40         INDIRIZZO DI BASE DEL PIA
                           ;
00000000:                  PIADDA   EQU      $0             OFFSET REG. DIREZIONE DATI A
00000000:                  PIADA    EQU      $0             OFFSET REG. DATI A
00000004:                  PIACA    EQU      $4             OFFSET REG. DI CONTROLLO A
00000002:                  PIADDB   EQU      $2             OFFSET REG. DIREZIONE DATI B
00000002:                  PIADB    EQU      $2             OFFSET REGISTRO DATI B
00000006:                  PIACB    EQU      $6             OFFSET REG. DI CONTROLLO B
                           ;
00000000:                  A_DATDIR EQU      $00            DAL LATO A TUTTI INGRESSI
00000004:                  A_CNTRL  EQU      04             SELEZIONA REGISTRO DATI
00000007:                  B_DATDIR EQU      $07            LE LINEE 0-2 DEL LATO B SONO USCITE
00000004:                  B_CNTRL  EQU      $04            SELEZIONA REGISTRO DATI
                           ;
00000007:                  COL_MASK EQU      $07            COLONNE COLLEGATE AI BIT 0-2
00000000:                  ROWGRND  EQU      $00            MASCHERA PER METTERE A MASSA TUTTE
                           ;                                LE RIGHE DELLA TASTIERA
                                    ORG      PROGRAM

00004000: 207C 0003
00004004: FF40             PGM13_5A MOVEA.L  #PIA,A0        PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004                 CLR.B    PIACA(A0)      INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                      MOVE.B   #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                      MOVE.B   #A_CNTRL,PIACA(A0)
00004016: 4228 0006                 CLR.B    PIACB(A0)      INIZIALIZZA IL LATO B
0000401A: 117C 0007
0000401E: 0002                      MOVE.B   #B_DATDIR,PIADDB(A0)
00004020: 117C 0004
00004024: 0006                      MOVE.B   #B_CNTRL,PIACB(A0)
                           ;
00004026: 1178 0000
0000402A: 0002                      MOVE.B   ROWGRND,PIADB(A0) A MASSA TUTTE LE RIGHE
                           ;
0000402C: 1028 0000        WAIT     MOVE.B   PIADA(A0),D0   PRENDI DATO DALLE COLONNE
00004030: 0200 0007                 ANDI.B   #COL_MASK,D0   MASCHERA I BIT DELLE COLONNE
00004034: 0C00 0007                 CMPI.B   #COL_MASK,D0   C'E' QUALCHE TASTO PREMUTO?
00004038: 67F2                      BEQ      WAIT           SE NO, ATTENDI
                           ;
0000403A: 4E75                      RTS

                                    END      PGM13_5A
```

Eliminando con una maschera i bit corrispondenti alle colonne, si evita qualsiasi problema dovuto dalle linee di input inutilizzate.

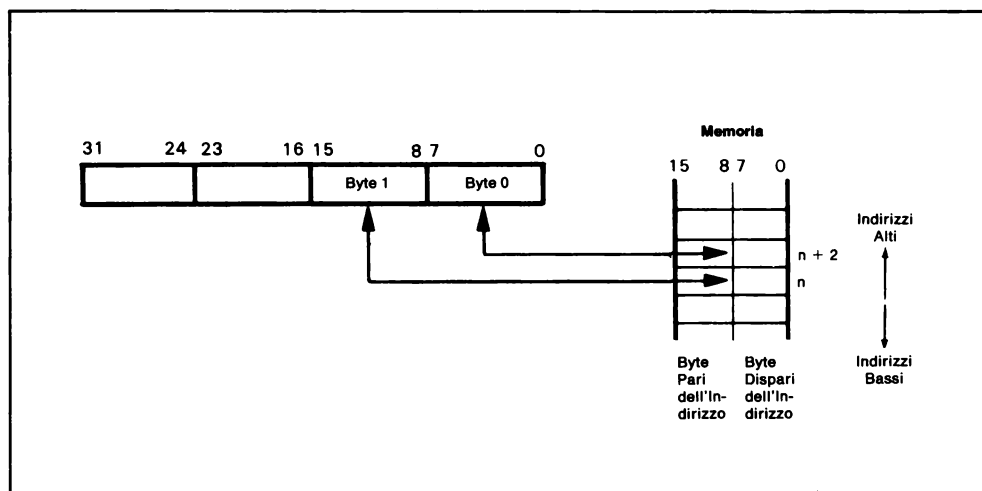
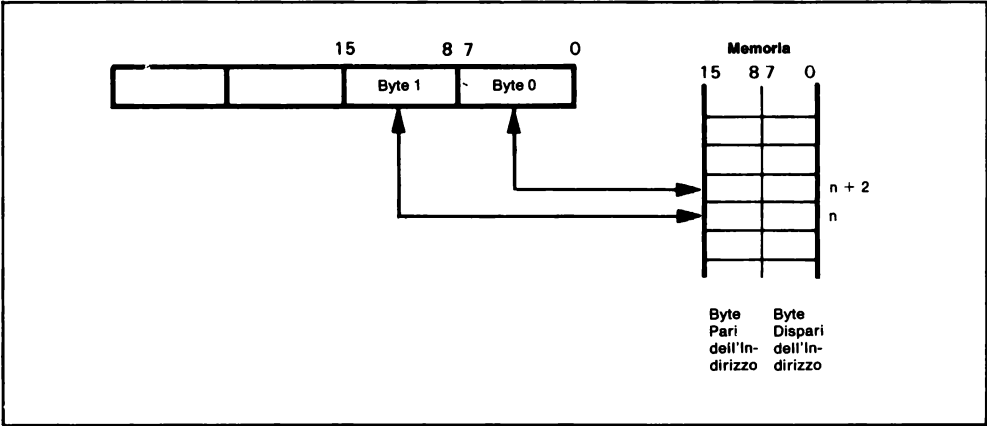
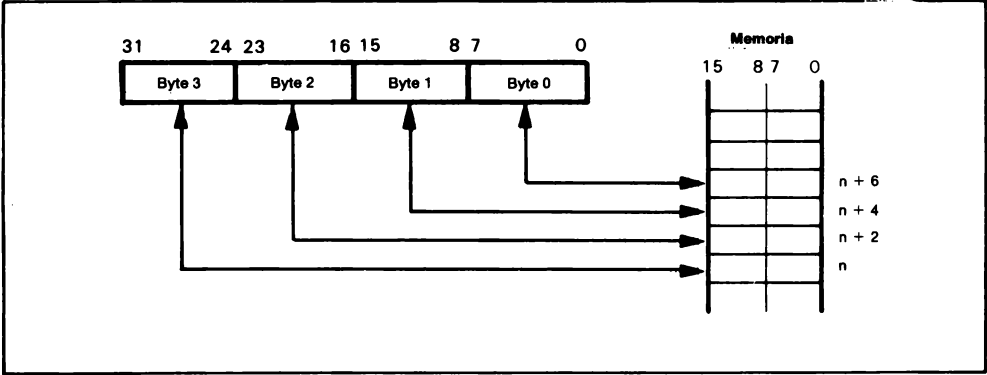
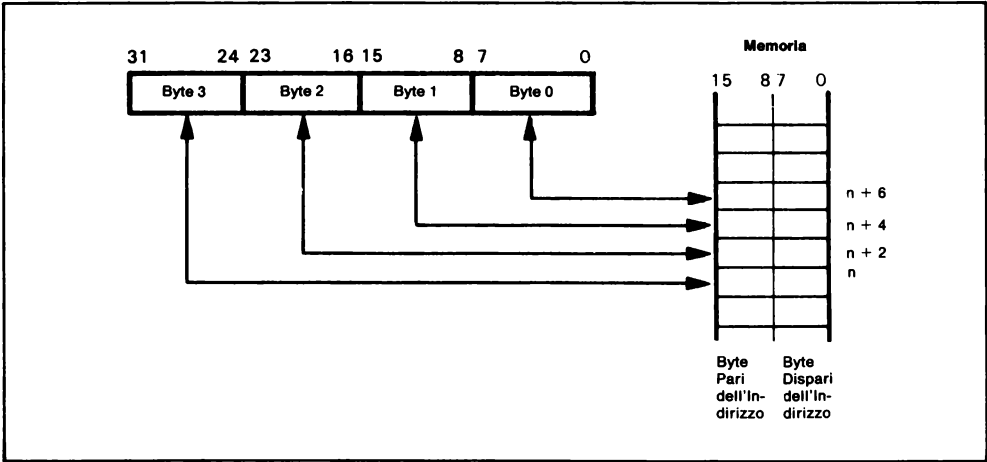
L'uso delle istruzioni RESET e MOVEP nelle Applicazioni di I/O

L'MC68000 impiega un I/O di tipo "memory-mapped"; si possono, cioè, indirizzare le periferiche come se fossero delle normali locazioni di memoria. Con questa tecnica, non sono necessarie delle speciali istruzioni di I/O. L'MC68000 dispone, comunque, di due istruzioni particolari, RESET e MOVEP, molto utili nelle operazioni di I/O.

Descrizione dell'istruzione RESET

L'istruzione RESET fa sì che il processore generi un impulso sul pin di reset. Essa agisce su ***tutte*** le periferiche collegate al segnale di reset dell'MC68000, senza alterare lo stato interno del processore, che si limita, semplicemente, ad eseguire l'istruzione successiva a quella di RESET. Si tratta di una delle cosiddette istruzioni speciali o "privilegiate" e può essere eseguita soltanto nel modo Supervisore, di cui parleremo più dettagliatamente nel Capitolo 15.

Descrizione dell'istruzione MOVEP

L'istruzione MOVEP (Move Peripheral) è simile alla normale istruzione MOVE, tranne per il fatto che il trasferimento di un dato avviene tra un registro dati e delle locazioni di memoria alternate. Ad esempio ecco ciò che accade quando deve essere trasferito un dato a 16 bit e l'indirizzo specificato è dispari; il movimento del dato è il seguente:

Per prima cosa vengono trasferiti i bit 8-15 del registro dati all'indirizzo iniziale specificato e, successivamente, tocca al byte di ordine basso che viene messo nella locazione corrispondente all'indirizzo iniziale + 2. Indicando un indirizzo pari, il trasferimento di un dato avviene nel modo seguente:

I trasferimenti di una long word (32 bit) avvengono allo stesso modo. Ecco un esempio con un indirizzo dispari:

Ecco un trasferimento a 32 bit con un indirizzo pari:

In ciascuno di questi casi, per primi vengono trasferiti il byte (o i byte) più significativi, mentre gli altri eventuali byte vengono messi nelle locazioni successive, incrementando di due l'indirizzo iniziale.

Uso dell'istruzione MOVEP

Fino a questo punto l'istruzione MOVEP potrebbe sembrare inutile; perchè si dovrebbero trasferire dei dati da e verso dei byte di memoria alternati, anzichè consecutivi? La risposta sta nel fatto che questa istruzione serve a trasferire dei dati fra l'MC68000 e dei dispositivi periferici ad 8 bit (come un PIA). Normalmente, le linee dati di un dispositivo di questo tipo sono collegate alle otto linee di ordine alto, o di ordine basso, del bus dati. Perciò, a causa del modo in cui il processore indirizza la memoria (ed un qualsiasi altro dispositivo collegato al bus di sistema), un dispositivo ad 8 bit sarà sempre collegato alla parte del bus dati con gli indirizzi dispari (bit 0-7) oppure a quella con gli indirizzi pari. Ora, **volendo trasferire dati formati da più di un byte da e verso un dispositivo ad 8 bit, dobbiamo assicurarci di utilizzare sempre la metà corretta del bus (la pari o la dispari). Con l'istruzione MOVEP, tutto questo diventa estremamente semplice.**

Descrizione del programma 13.5b

Il Programma 13-5b utilizza entrambe le istruzioni, RESET e MOVEP per realizzare una funzione analoga a quella del Programma 13-5a. L'istruzione RESET provoca il reset del PIA (e, naturalmente, di tutti gli altri dispositivi collegati alla linea di reset), selezionando il registro direzione dati e configurando tutte le linee dati come ingressi. Le istruzioni MOVEP provvedono, poi, a selezionare i due registri dati del PIA.

Programma 13-5b

```
00004000:                   PROGRAM  EQU      $4000
00006000:                   DATA     EQU      $6000
                            ;
0003FF40:                   PIA      EQU      $3FF40          INDIRIZZO DI BASE DEL PIA
                            ;
00000000:                   PIADDA   EQU      $0              OFFSET REG. DIREZIONE DATI A
00000000:                   PIADA    EQU      $0              OFFSET REG. DATI A
00000004:                   PIACA    EQU      $4              OFFSET REG. DI CONTROLLO A
00000002:                   PIADDB   EQU      $2              OFFSET REG. DIREZIONE DATI B
00000002:                   PIADB    EQU      $2              OFFSET REGISTRO DATI B
00000006:                   PIACB    EQU      $6              OFFSET REG. DI CONTROLLO B
                            ;
00000007:                   B_DATDIR EQU      $07             LE LINEE 0-2 DEL LATO B SONO USCITE
00000404:                   AB_CNTRL EQU      $0404           SELEZIONA REGISTRI DATI A & B
                            ;
00000007:                   COL_MASK EQU      $07             MASCHERA PER I BIT DEL DATO
00000000:                   ROWGRND  EQU      $00             MASCHERA PER METTERE A MASSA TUTTE
                            ;                                 LE RIGHE DELLA TASTIERA
                                     ORG      PROGRAM

00004000: 4E70              PGM13_5B RESET                    RESET DI TUTTI I DISPOSITIVI EST.
00004002: 207C 0003
00004006: FF40                       MOVEA.L #PIA,A0          INDIRIZZO DEL PIA
00004008: 117C 0007
0000400C: 0002                       MOVE.B  #B_DATDIR,PIADDB(A0) DAL LATO B TUTTE USCITE
0000400E: 303C 0404                  MOVE.W  #AB_CNTRL,D0     WORD DI CONTROLLO AD ENTRAMBI
00004012: 0188 0004                  MOVEP.W D0,PIACA(A0)     ...I REG. DI CONTROLLO DEL PIA
                            ;
00004016: 117C 0000
0000401A: 0002                       MOVE.B  #ROWGRND,PIADB(A0) A MASSA TUTTE LE RIGHE
                            ;
0000401C: 1028 0000         WAIT     MOVE.B  PIADA(A0),D0     PRENDI DATO DALLE COLONNE
00004020: 0200 0007                  ANDI.B  #COL_MASK,D0     MASCHERA I BIT DELLE COLONNE
00004024: 0C00 0007                  CMPI.B  #COL_MASK,D0     C'E' QUALCHE TASTO PREMUTO?
00004028: 67F2                       BEQ     WAIT             SE NO, ATTENDI
                            ;
0000402A: 4E75                       RTS

                                     END     PGM13_5B
```

Naturalmente, per una tastiera 3 x 3 o 4 x 4 è sufficiente una sola porta. Modificate il programma in modo da utilizzare soltanto la porta A.

Funzione 13-5b. Identificare un Tasto.

Scopo: Identificare la chiusura di un tasto, mettendo il numero corrispondente nel registro dati D0.

La procedura è la seguente:

1 Porre a 1 il numero del tasto e tutti i bit della porta di output della tastiera, tranne il bit 0, che deve, invece, essere azzerato.
2. Prelevare gli input delle colonne, leggendo la porta di input.
3. Se l'input relativo ad una delle colonne è zero, passare alla Fase 7.
4. Sommare il numero di colonne al numero del tasto per ottenere la riga successiva.
5. Aggiornare il contenuto della porta di output, ruotando a sinistra di una posizione il contenuto del registro dati.
6 Decrementare il contatore di riga. Se tutte le righe non sono state controllate, ritornare alla Fase 2, altrimenti passare alla Fase 9.
7. Aggiungere 1 al numero del tasto. Eseguire lo shift a destra di una posizione degli input di colonna.
8. Se il Carry = 1, ritornare alla Fase 7.
9. Fine del programma.

Questo programma non attende che l'operatore prema un tasto, perciò il tasto deve essere premuto prima dell'esecuzione del programma. Quali modifiche sono necessarie perchè il programma attenda che venga premuto almeno un tasto?

Programma 13-5c

```
00004000:          PROGRAM   EQU    $4000
00006000:          DATA      EQU    $6000
                   ;
0003FF40:          PIA       EQU    $3FF40      INDIRIZZO DI BASE DEL PIA
                   ;
00000000:          PIADDA    EQU    $0          OFFSET REG. DIREZIONE DATI A
00000000:          PIADA     EQU    $0          OFFSET REG. DATI A
00000004:          PIACA     EQU    $4          OFFSET REG. DI CONTROLLO A
00000002:          PIADDB    EQU    $2          OFFSET REG. DIREZIONE DATI B
00000002:          PIADB     EQU    $2          OFFSET REGISTRO DATI B
00000006:          PIACB     EQU    $6          OFFSET REG. DI CONTROLLO B
                   ;
00000000:          A_DATDIR  EQU    $00         DAL LATO A TUTTI INGRESSI
00000007:          B_DATDIR  EQU    $07         LE LINEE 0-2 DEL LATO B SONO USCITE
00000004:          A_CNTRL   EQU    $04         SELEZIONA REGISTRO DATI
00000004:          B_CNTRL   EQU    $04         SELEZIONA REGISTRO DATI
                   ;
00000007:          COL_MASK  EQU    $07         COLONNE COLLEGATE AI BIT 0-2
00000003:          COLCOUNT  EQU    $03         NUM. DI COLONNE SULLA TASTIERA
00000003:          ROWCOUNT  EQU    $03         NUM. DI RIGHE SULLA TASTIERA
000000FE:          ROWGRND   EQU    $FE         MASCHERA PER METTERE A MASSA LA
                                                PRIMA RIGA
                   ;
                             ORG    PROGRAM
```

```
00004000: 207C 0003
00004004: FF40           PGM13_5C MOVEA.L #PIA,A0           PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004               CLR.B   PIACA(A0)         INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                    MOVE.B  #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                    MOVE.B  #A_CNTRL,PIACA(A0)
00004016: 4228 0006               CLR.B   PIACB(A0)         INIZIALIZZA IL LATO B
0000401A: 117C 0007
0000401E: 0002                    MOVE.B  #B_DATDIR,PIADDB(A0)
00004020: 117C 0004
00004024: 0006                    MOVE.B  #B_CNTRL,PIACB(A0)
                         ;
00004026: 163C 00FE               MOVE.B  #ROWGRND,D3       PRENDI MASCHERA COLLEG. A MASSA
0000402A: 70FF                    MOVEQ   #-1,D0            CONTATORE NUM. TASTO
0000402C: 7202                    MOVEQ   #ROWCOUNT-1,D1    CORREGGI CONT.RIGHE PER DBRA
                         ;
0000402E: 1143 0002      ROW      MOVE.B  D3,PIADB(A0)      A MASSA UNA RIGA
00004032: 1428 0000               MOVE.B  PIADA(A0),D2      PRENDI I BIT DI COLONNA
00004036: 0202 0007               ANDI.B  #COL_MASK,D2      MASCHERA BIT DI COLONNA
0000403A: 0C02 0007               CMPI.B  #COL_MASK,D2      UN TASTO PREMUTO IN QUESTA RIGA?
0000403E: 660C                    BNE.S   COLUMN            ...ALLORA CONTROLLA COLONNE
                         ;
00004040: 5600                    ADDQ.B  #COLCOUNT,D0      ...ALTRIMENTI RIGA SEGUENTE
00004042: E31B                    ROL.B   #1,D3             AGGIORNA MASCHERA
00004044: 51C9 FFE8               DBRA    D1,ROW            CONTINUA CON LE ALTRE RIGHE
                         ;
00004048: 70FF                    MOVEQ   #-1,D0            CODICE PER NESSUN TASTO PREMUTO
0000404A: 6006                    BRA.S   DONE              FINE, RITORNA CON -1 IN D0
                         ;
0000404C: 5200           COLUMN   ADDQ.B  #1,D0             NUM. TASTO := NUM. TASTO + 1
0000404E: E20A                    LSR.B   #1,D2             QUESTA COLONNA E' A MASSA?
00004050: 65FA                    BCS     COLUMN
                         ;
00004052: 4E75           DONE     RTS

                                  END     PGM13_5C
```

Ogni volta che l'esplorazione di una riga dà esito negativo dobbiamo sommare il numero delle colonne (3 nel nostro esempio) al numero del tasto per spostarci alla riga successiva (provate questa procedura sulla tastiera di Figura 13-12).

Qual è il valore presente nel registro dati D0, se non viene premuto nessun tasto? Modificate il programma, in modo che, in tal caso, esso dia inizio ad una nuova esplorazione.

Una soluzione alternativa è quella di sfruttare la bidirezionalità del PIA.[7] La procedura sarebbe:

1. Collegare a massa tutte le colonne e salvare gli input di riga.
2. Collegare a massa tutte le righe e salvare tutti gli input di colonna.
3. Utilizzare tutti gli input di riga e di colonna per stabilire il numero del tasto tramite una tabella.

Scrivere un programma che utilizza questa procedura.

Diagramma di Flusso 13-5c

13-6. Una tastiera codificata

Il processore preleva dei dati, quando sono disponibili, da una tastiera codificata che, in corrispondenza del trasferimento di un dato, provvede ad inviare un segnale di strobe.

Caratteristiche di una tastiera codificata

Una tastiera codificata fornisce un codice diverso per ogni tasto e possiede dei circuiti interni, che eseguono automaticamente le procedure di esplorazione e identificazione descritte nell'esempio precedente. Il software necessario in questo caso è molto più semplice di quello richiesto da una tastiera non codificata, che, però, ha il vantaggio di essere meno costosa.

Le tastiere codificate utilizzano matrici di diodi e codificatori TTL o MOS. I codici sono di tipo ASCII, EBCDIC oppure realizzati su misura. Spesso i circuiti di codifica contengono anche delle PROM.

Oltre a fornire il codice corrispondente al tasto premuto, un circuito di codifica provvede anche ad eliminare i rimbalzi dei tasti ed a gestire il "rollover", cioè il problema dovuto alla pressione contemporanea di più tasti. Le modalità più diffuse per la gestione di questo fenomeno sono il "rollover a 2 tasti", in cui due tasti (ma non di più) premuti nello stesso momento sono considerati come due chiusure distinte, ed il "rollover ad n tasti", in cui un qualsiasi numero di tasti premuti contemporaneamente vengono interpretati come chiusure separate.

Una tastiera codificata genera anche un segnale di strobe in corrispondenza del trasferimento di un dato, per indicare che è stato premuto un altro tasto. La Figura 13-13 mostra l'interfaccia fra una tastiera codificata ed il microprocessore MC68000. Colleghiamo la linea di strobe della tastiera all'ingresso CA1, in modo che un impulso sulla linea di strobe ponga ad uno il bit 7 del registro di controllo del PIA. Il bit 1 di questo stesso registro determina, a sua volta, quale dei due fronti di un impulso (quello di salita o quello di discesa) sarà riconosciuto dal PIA. Se il bit 1 è uguale a 0 sarà il fronte di discesa (transizione alto-basso), se è uguale a 1 il fronte di salita (transizione basso-alto).

Il PIA, oltre ad una porta dati parallela, contiene anche una porta di stato seriale sensibile al fronte di un impulso e fornita di latch ed un inverter che gli permette di distinguere il tipo di fronte. Perciò, un PIA può sostituire molti degli elementi normalmente presenti in un circuito, come flip-flop, gate, inverter e buffer. Questo rappresenta un grosso vantaggio in fase di progettazione, perchè ci consente di apportare, in ogni momento, delle modifiche, cambiando solamente il contenuto del registro di controllo, senza dover modificare l'hardware. Ad esempio, per cambiare il fronte attivo sulla linea di strobe, basta cambiare un solo bit del programma, senza l'aggiunta di nuove componenti hardware o la necessità di nuovi collegamenti.

Ricordate, comunque, che un PIA non contiene un latch di input, che sarebbe indispensabile qualora la tastiera non fosse in grado di mantenere stabile un dato per un tempo sufficiente. Anche questo latch può essere controllato dal segnale di strobe.

Funzione: Attendere un segnale di strobe attivo basso sulla linea di controllo CA1 e, quindi, caricare il dato della tastiera nel registro dati D0.

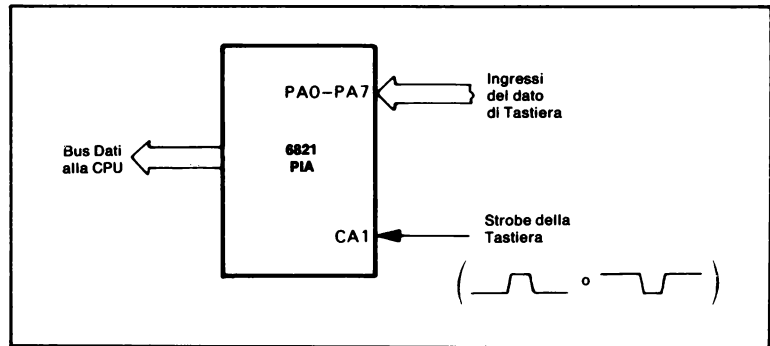
Figura 13-13. Interfaccia di I/O per una Tastiera Codificata.

Un particolare circuito, all'interno del PIA, fa sì che un'operazione di lettura del registro dati azzeri il bit di stato.

Diagramma di Flusso 13-6

Notate la "falsa" lettura in fase di inizializzazione. La sua funzione è quella di azzerare il bit 7 del registro di controllo A, in modo da essere sicuri che una sua accidentale attivazione non crei dei problemi.

Programma 13-6

```
00004000:                 PROGRAM  EQU      $4000
00006000:                 DATA     EQU      $6000
                          ;
0003FF40:                 PIA      EQU      $3FF40          INDIRIZZO DI BASE DEL PIA
                          ;
00000000:                 PIADDA   EQU      $0              OFFSET REG. DIREZIONE DATI A
00000000:                 PIADA    EQU      $0              OFFSET REG. DATI A
00000004:                 PIACA    EQU      $4              OFFSET REG. DI CONTROLLO A
                          ;
00000000:                 A_DATDIR EQU      $00             DAL LATO A SONO TUTTI INGRESSI
00000004:                 A_CNTRL  EQU      $04             SELEZIONA REGISTRO DATI. USA CA1
                          ;                                 COME INPUT
                                   ORG      PROGRAM

00004000: 207C 0003
00004004: FF40            PGM13_6  MOVEA.L  #PIA,A0         PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004                CLR.B    PIACA(A0)       INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                     MOVE.B   #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                     MOVE.B   #A_CNTRL,PIACA(A0)
00004016: 1028 0000                MOVE.B   PIADA(A0),D0    FALSA LETTURA-AZZERA I BIT DI STATO
                          ;
0000401A: 1028 0004       WAIT     MOVE.B   PIACA(A0),D0    E' STATO PREMUTO UN TASTO/
0000401E: 6AFA                     BPL      WAIT            SE NO, ATTENDI
00004020: 1028 0000                MOVE.B   PIADA(A0),D0    ..ALTRIMENTI PRELEVA UN DATO
                          ;                                 DALLA TASTIERA
00004024: 4E75                     RTS

                                   END      PGM13_6
```

Se vogliamo che il bit 7 del registro di controllo sia attivato dalla presenza di transizioni basso-alto sulla linea di strobe della tastiera, basta sostituire A_CNTRL EQU $04 con A_CNTRL EQU $06.

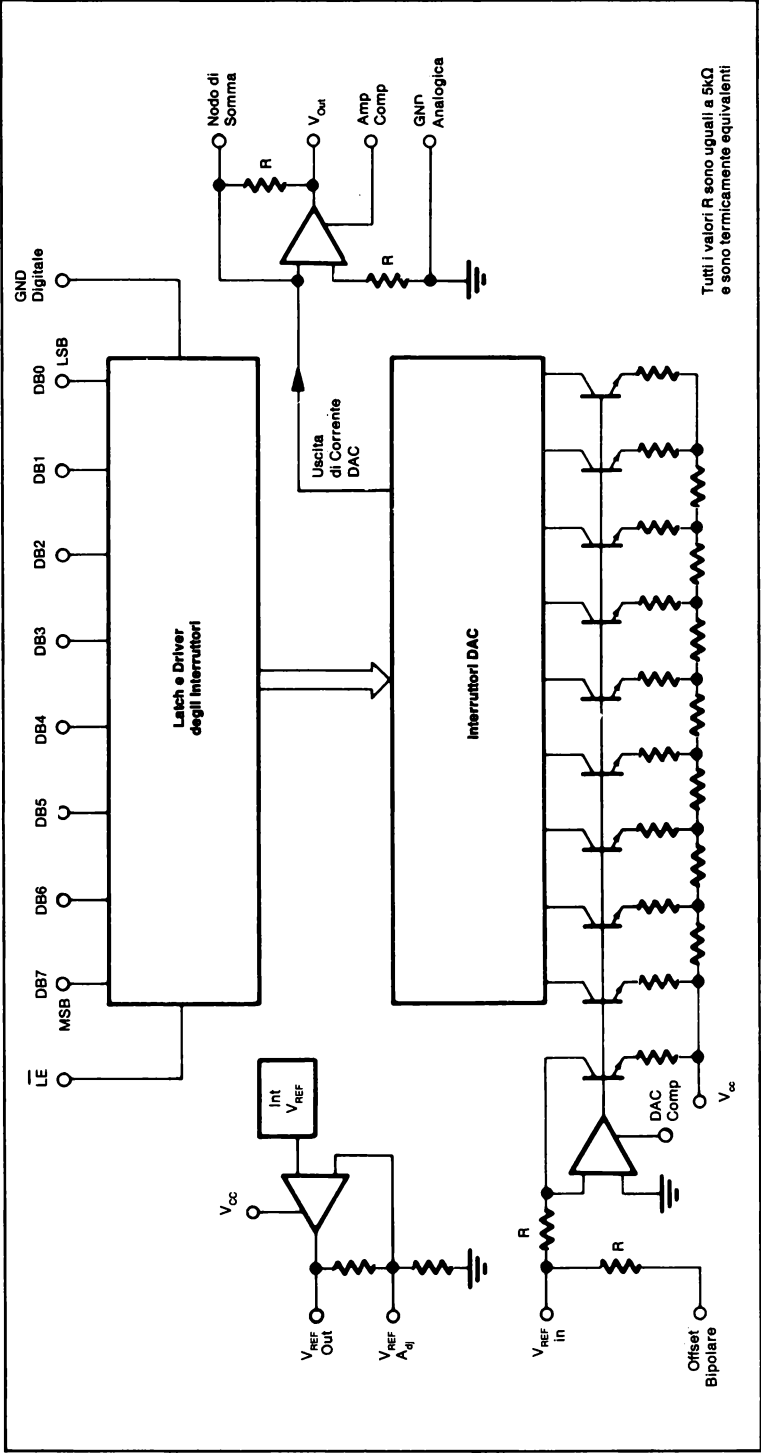
Se collegassimo la linea di strobe della tastiera a CA2 allora come latch di stato sarebbe utilizzato il bit 6 del registro di controllo.

Dimostrate che la lettura del registro dati azzera il bit di stato, indicando che la CPU ha letto il dato e permettendo il verificarsi della successiva operazione di input. Un suggerimento: salvate il contenuto del registro di controllo del PIA in memoria, prima e dopo l'esecuzione dell'istruzione MOVE.B D0, PIADA(A0). Cosa accadrebbe sostituendo MOVE.B PIADA(A0),DO con MOVE.B D0,PIADA(A0)? Non dimenticate che un'operazione di scrittura del registro dati non azzera il bit di stato e questo non accade nemmeno scrivendo o leggendo un dato dal registro di controllo. Cosa accade sostituendo MOVE.B PIADA(A0),D0 con MOVE.B PIACA(A0),DO oppure con MOVE.B D0,PIACA(A0)?

Conoscere nei dettagli gli effetti che le varie istruzioni hanno sui registri del PIA può rivelarsi utile, volendo impiegare le linee di controllo per scopi che non hanno nulla a che vedere con le porte dati. Ad esempio, se utilizziamo un PIA per interfacciare una periferica molto semplice (ad es. un gruppo di interruttori o di LED) che non richiede l'impiego delle linee di controllo, queste le potremo usare come linee seriali di I/O senza la necessità di hardware aggiuntivo. Il solo problema è quello di doverci servire di un dispositivo che è stato realizzato presupponendo una stretta connessione fra queste linee, utilizzate adesso per un I/O seriale, e la porta dati parallela.

13-7. Un convertitore digitale-analogico

Il processore invia dei dati ad un convertitore digitale-analogico ad 8 bit, che ha un segnale Latch Enable attivo basso.

Figura 13-14. Il Convertiore D/A NE 5018 della Signetics.

Tipi di convertitori A/D

I convertitori digitali-analogici che generano segnali continui necessari per i solenoidi, i relè, gli attuatori ed altri dispositivi elettrici o meccanici sono costituiti da interruttori e reti di resistenze di opportuno valore, cui basta fornire una tensione di riferimento ed aggiungere qualche altro circuito analogico o digitale. Attualmente, sono disponibili anche unità complete, a prezzi accessibili.

La Figura 13-14 mostra il convertitore D/A ad 8 bit NE5018 della Signetics, che contiene un latch di input parallelo ad 8 bit. Un livello basso sull'ingresso LE (Latch Enable) fa entrare il dato di ingresso nei latch, dove rimane dopo che LE è divenuto alto.

Interfaccia per un Convertitore D/A

La Figura 13-15 mostra l'interfacciamento di un NE5018 con un sistema MC68000. La porta B del PIA genera automaticamente un segnale Latch Enable attivo basso su CB2, che agisce come una linea OUTPUT READY, segnalando l'invio di un dato alla porta di output. Nel modo operativo in cui il segnale è costituito da un breve impulso, CB2 diventa bassa automaticamente in corrispondenza dell'impulso di clock successivo ad un'operazione di scrittura nel registro dati B e resta bassa fino all'impulso di clock successivo (cfr. Tabella 13-5).

I bit del registro di controllo che permettono al PIA di funzionare in questo modo sono:

Bit 5 = 1 per rendere CB2 un uscita
Bit 4 = 0 per avere un impulso su CB2
Bit 3 = 1 perchè la durata dell'impulso deve essere pari ad un ciclo di clock

Il PIA contiene un latch di output, per cui un dato rimane stabile durante e dopo la conversione, anche se rimane sul bus dati solo per la durata di un ciclo di clock. I latch di output sono indispensabili

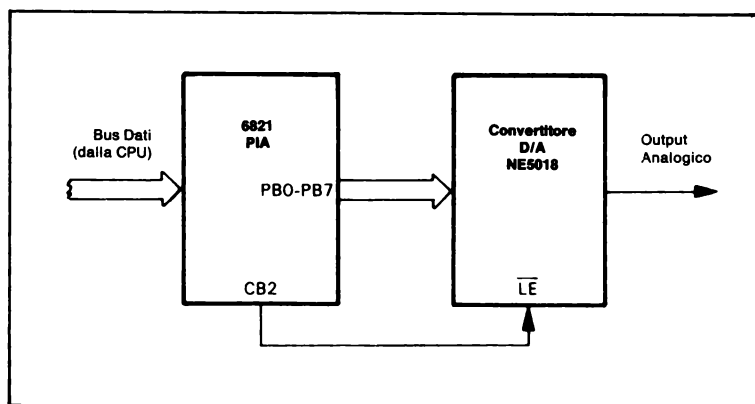



Figura 13-15. Interfaccia per un Convertitore Digitale-Analogico ad 8 bit.

nei microcomputer in quanto il processore utilizza continuamente il bus dati per trasferire dati e istruzioni da e verso la memoria. In genere, un convertitore impiega solo pochi microsecondi per generare degli output analogici, ma ci sono altre periferiche che devono disporre di un dato per un periodo di tempo molto più lungo.

Quando otto bit non sono sufficienti bisogna utilizzare i convertitori da 10 fino a 16 bit, che oggi sono ormai largamente diffusi. Quelli indicati come "compatibili con i microprocessori" dispongono, di solito, di porte dati distinte per i byte più e meno significativi. Dispositivi di questo tipo sono molto più facili da interfacciare rispetto a quelli che accettano tutto il dato contemporaneamente attraverso un'unica porta.

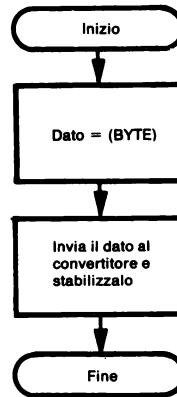
Un PIA funziona, in questo caso, sia come porta dati parallela che come porta di controllo seriale. Dopo che la CPU ha messo un dato nella porta di output, CB2 fornisce un impulso della durata di un ciclo di clock sufficiente per un convertitore come l'NE5018, che richiede impulsi di 400ns.

Funzione 13-7. Inviare un Dato ad un Convertitore D/A

Scopo: Inviare il dato contenuto nella variabile BYTE, alla locazione di memoria 6000, ad un convertitore D/A.

Programma 13-7a

```
00004000:                    PROGRAM  EQU      $4000
00006000:                    DATA     EQU      $6000
                             ;
0003FF40:                    PIA      EQU      $3FF40        INDIRIZZO DI BASE DEL PIA
                             ;
00000002:                    PIADDB   EQU      $2            OFFSET REG. DIREZIONE DATI B
00000002:                    PIADB    EQU      $2            OFFSET REGISTRO DATI B
00000006:                    PIACB    EQU      $6            OFFSET REG. DI CONTROLLO B
                             ;
000000FF:                    B_DATDIR EQU      $FF           LE LINEE DEL LATO B SONO USCITE
0000002C:                    B_CNTRL  EQU      $2C           SELEZIONA REGISTRO DATI ED UNO
                                                             STROBE SU CB2
                             ;
                                      ORG      DATA

00006000:                    BYTE     DS.B     1             DATO DA INVIARE AD UN CONV. D/A
                             ;
                                      ORG      PROGRAM

00004000: 207C 0003
00004004: FF40               PGM13_7A MOVEA.L  #PIA,A0       PRENDI IND. DI BASE DEL PIA
00004006: 4228 0006                   CLR.B    PIACB(A0)     INIZIALIZZA IL LATO B
0000400A: 117C 00FF
0000400E: 0002                        MOVE.B   #B_DATDIR,PIADDB(A0)
00004010: 117C 002C
00004014: 0006                        MOVE.B   #B_CNTRL,PIACB(A0)
                             ;
00004016: 1178 6000
0000401A: 0002                        MOVE.B   BYTE,PIADB(A0)  INVIA DATO A CONV.D/A E LATCH
                             ;
0000401C: 4E75                        RTS

                                      END      PGM13_7A.
```

Diagramma di Flusso 13-7a

Il PIA genera automaticamente un segnale su CB2 dopo che la CPU ha messo un valore nel registro dati senza che vi sia bisogno di nessuna particolare istruzione. Sebbene operazioni automatiche come questa facciano risparmiare tempo e memoria, spesso causano anche degli incovenienti, se non ne indichiamo la presenza nel campo riservato ai commenti o nella documentazione allegata al programma. Per comprendere il funzionamento di questa interfaccia, è necessario sapere come funziona il 6821 e poter disporre di uno schema dell'hardware e del listato del programma. Ciò rende particolarmente difficili sia la manutenzione che l'inserimento di eventuali modifiche.

L'impulso ha una durata pari ad un ciclo di clock. Se questo non è abbastanza (o se è necessario un impulso attivo alto), possiamo utilizzare un livello su CB2. Questo modo operativo è detto manuale, in quanto il PIA non genera automaticamente un impulso. Ecco il relativo programma:

```
00004000:                  PROGRAM  EQU      $4000
00006000:                  DATA     EQU      $6000
                           ;
0003FF40:                  PIA      EQU      $3FF40         INDIRIZZO DI BASE DEL PIA
                           ;
00000002:                  PIADDB   EQU      $2             OFFSET REG. DIREZIONE DATI B
00000002:                  PIADB    EQU      $2             OFFSET REGISTRO DATI B
00000006:                  PIACB    EQU      $6             OFFSET REG. DI CONTROLLO B
                           ;
000000FF:                  B_DATDIR EQU      $FF            LE LINEE DEL LATO B SONO USCITE
00000034:                  B_CNTRL  EQU      $34            SELEZIONA REGISTRO DATI,
                           ;*                               STROBE "MANUALE" SU CB2
00000003:                  DASTABIT EQU      $03            CB2 COLLEGATO A LATCH ENABLE
                           ;
                                    ORG      DATA

00006000:                  BYTE     DS.B     1              DATO DA INVIARE AD UN CONV. D/A
                           ;
                                    ORG      PROGRAM

00004000: 207C 0003
00004004: FF40             PGM13_7B MOVEA.L  #PIA,A0        PRENDI IND. DI BASE DEL PIA
00004006: 4228 0006                 CLR.B    PIACB(A0)      INIZIALIZZA IL LATO B
0000400A: 117C 00FF
0000400E: 0002                      MOVE.B   #B_DATDIR,PIADDB(A0)
00004010: 117C 0034
00004014: 0006                      MOVE.B   #B_CNTRL,PIACB(A0)
```

```
00004016: 1178 6000         ;
0000401A: 0002                  MOVE.B  BYTE,PIADB(A0)  INVIA DATO A CONV.D/A E LATCH
0000401C: 08E8 0003
00004020: 0006                  BSET    #DASTABIT,PIACB(A0)  APRI LATCH CONV. D/A
00004022: 08A8 0003
00004026: 0006                  BCLR    #DASTABIT,PIACB(A0)  LATCH DEL DATO
                            ;
00004028: 4E75                  RTS
                                END     PGM13_7B
```

Questa soluzione richiede un maggior numero di istruzioni, ma fornisce un impulso di maggior durata ed è più comprensibile. In questo caso il bit 4 del registro di controllo del PIA è posto a uno per avere su CB2 un livello del valore indicato dal bit 3, che provvederemo a mettere a uno o ad azzerare con le istruzioni BSET e BCLR.

Nel modo di livello o manuale CB2 è completamente indipendente dalle operazioni che avvengono sulla porta dati parallela e fornisce un semplice output seriale disponibile per un qualsiasi impiego. La sola precauzione da prendere è quella di non cambiare gli altri bit del registro di controllo del PIA, dal momento che hanno delle funzioni del tutto diverse. Per questo ci serviamo delle istruzioni BTST e BCLR, che agiscono selettivamente sul bit 3 del registro di controllo senza modificare gli altri.

13-8. Convertitore analogico-digitale

Il processore preleva un dato da un convertitore analogico-digitale ad 8 bit che richiede un impulso per dare inizio al processo di conversione e fornisce un segnale di fine conversione per indicarne il completamento e la disponibilità di un dato valido.

I convertitori analogico-digitali servono a convertire i segnali continui generati da vari tipi di sensori e trasduttori in input digitali destinati agli elaboratori.

Un tipo di convertitore analogico-digitale è quello ad approssimazione successiva, che effettua un confronto diretto di un bit durante ogni ciclo di clock: è molto veloce, ma ha una scarsa immunità al rumore. Un altro tipo di convertitore analogico-digitale è quello ad integrazione a doppia pendenza. Questi dispositivi sono più lenti, ma anche più insensibili al rumore. Sono impiegate anche altre tecniche di conversione, come quella a bilanciamento di carica incrementale.

In genere, questi convertitori hanno bisogno di alcuni circuiti esterni analogici o digitali, ma sono disponibili anche delle unità complete a basso prezzo.

La Figura 13-16 contiene la descrizione ed il diagramma di temporizzazione del convertitore A/D MM5357 della National, che contiene dei latch di output e delle uscite dati tri-state. Un impulso sulla linea di inizio conversione (STRT CONV) dà, appunto, inizio al processo di conversione dell'input analogico. Dopo circa 40 cicli di

clock (il convertitore richiede un clock di livello TTL con una durata minima dell'impulso pari a 400 ns.), il risultato andrà nei latch di output, mentre l'uscita EOC indicherà la fine della conversione, diventando alta. Il dato viene letto dai latch, mettendo un '1' sull'ingresso OUTPUT ENABLE. La Figura 13-17 mostra i collegamenti di questo dispositivo ed un tipico circuito applicativo.

Convertitore A/D MM5357 ad 8 bit della NATIONAL

Descrizione Generale

L'MM5357 è un convertitore A/D monolitico ad 8 bit realizzato con tecnologia MOS. Contiene un comparatore ad alta impedenza, 256 resistori e switches analogici, logica di controllo e stabilizzatori di output. La conversione è eseguita utilizzando la tecnica delle approssimazioni successive. L'output binario è a tre stati per permettere il convogliamento nelle normali linee dati.

Caratteristiche

- Basso costo
- Range di input + 5 V, 10 V
- Output a tre stati
- Contiene stabilizzatori di output
- Compatibile TTL

Specificazioni chiave

Risoluzione	8 bit
Linearità	+ 1/2 LSB
Velocità di conversione	40 µs
Impedenza di input	> 100 MΦ
Voltaggi	+ 5 V, - 12 V, GND
Range del clock	da 5 khz a 2.0 MHz

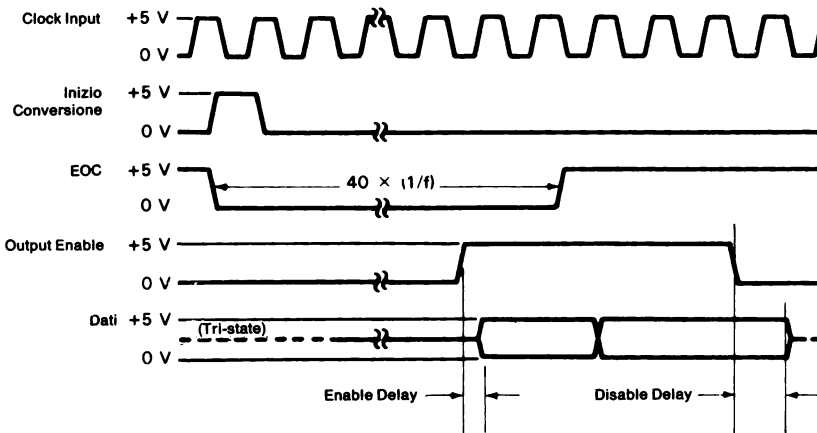
Diagramma di Temporizzazione:

I Dati sono valori binari complementari (Il valore max di output è costituito da tutti "0")

Figura 13-16. Descrizione Generale e Diagramma di Temporizzazione del Convertitore A/D 5357 della National.

Interfaccia per un Convertitore A/D

La Figura 13-18 mostra l'interfaccia fra un microprocessore MC68000 ed il convertitore A/D 5357. La linea di controllo CA2 è

impiegata nel modo livello (o manuale) per avere un impulso attivo basso di inizio conversione sufficientemente lungo. Il segnale di fine conversione è collegato alla linea di controllo CA1, per cui, quando la linea EOC diventa alta, il bit 7 del registro di controllo A del PIA è posto a uno. La transizione attiva sulla linea di fine conversione è il fronte di salita (basso-alto) che indica, appunto, il completamento del processo di conversione.

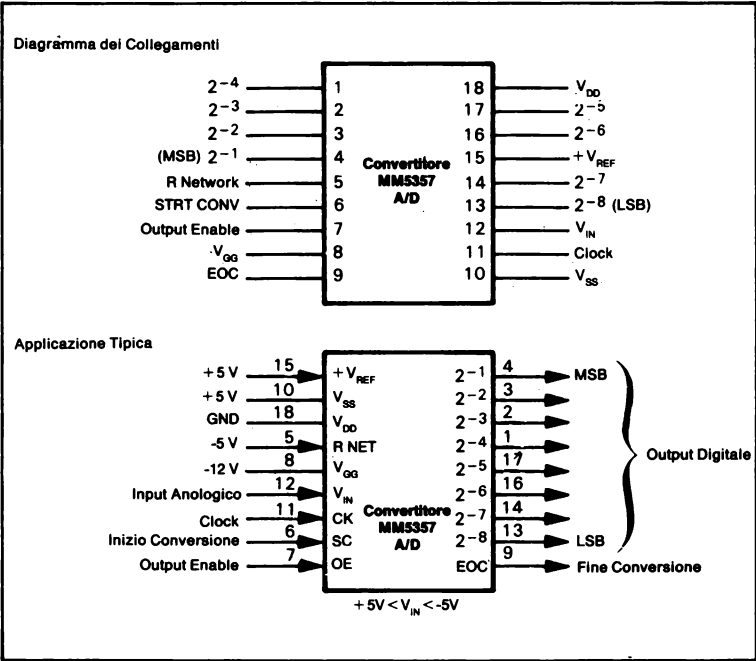



Figura 13-17. Diagramma dei Collegamenti ed una Tipica Applicazione di un Convertitore A/D 5357 della National.

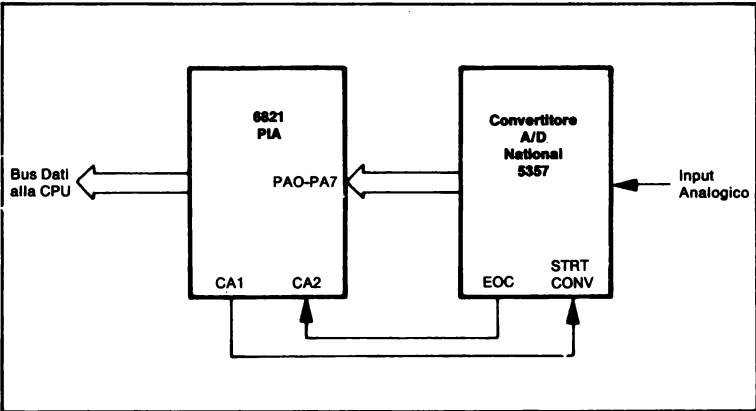
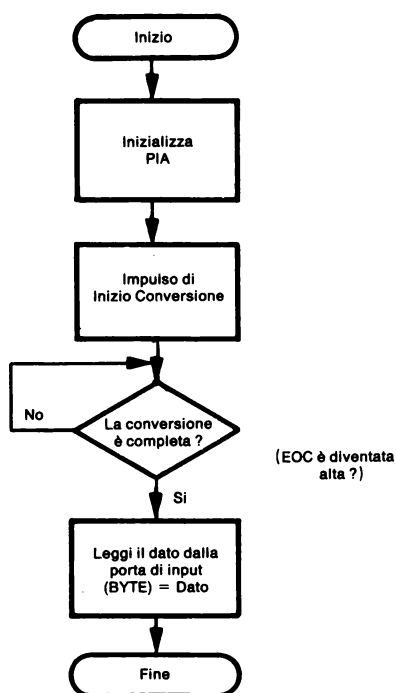



Figura 13-18. Interfaccia per un Convertitore Analogico-digitale ad 8 bit.

In questo caso, utilizziamo il PIA 6821 per gestire un input parallelo, un input seriale ed un output seriale, dato che un convertitore A/D ha bisogno di un handshake completo. Il pin OUTPUT ENABLE del convertitore è collegato ad una tensione positiva, poichè non mettiamo direttamente il dato sul bus dati tri-state del microprocessore. Come potete osservare nella Figura 13-6, i dati provenienti dal convertitore sono valori binari complementari (un output costituito da tutti zeri è il valore massimo).

Funzione 13-8. Input da un Convertitore

Scopo: Iniziare il processo di conversione, attendere che la linea di fine conversione (EOC) diventi alta, quindi leggere il dato e memorizzarlo nella variabile BYTE, alla locazione 6000.

Diagramma di Flusso 13-8

I bit del registro di controllo del PIA hanno i seguenti valori:
Bit 5 = 1 per configurare CA2 come uscita
Bit 4 = 1 per avere un livello su CA2 (cioè, per operare nel modo manuale).

Bit 3 = 0 perchè la linea di inizio conversione deve, inizialmente, essere bassa.

Bit 1 = 1 perchè il bit 7 deve essere attivato da una transizione basso-alto (fronte di salita) sulla linea di fine conversione.

Invece di controllare il bit di stato potremmo far ricorso ad una routine di ritardo, la cui durata sia maggiore del tempo richiesto dal processo di conversione.

Programma 13-8:

```
00004000:                      PROGRAM  EQU      $4000
00006000:                      DATA     EQU      $6000
                               ;
0003FF40:                      PIA      EQU      $3FF40          INDIRIZZO DI BASE DEL PIA
                               ;
00000000:                      PIADDA   EQU      $0              OFFSET REG. DIREZIONE DATI A
00000000:                      PIADA    EQU      $0              OFFSET REG. DATI A
00000004:                      PIACA    EQU      $4              OFFSET REG. DI CONTROLLO A
                               ;
00000000:                      A_DATDIR EQU      $00             DAL LATO A SONO TUTTI INGRESSI
00000036:                      A_CNTRL  EQU      $36             CA2 USCITA BASSA, SEL. REG. DATI
                               ; *                               CA1 ATTIVO SU TRANS. BASSO-ALTO
00000003:                      ADSTABIT EQU      $03             CA2 A LINEA INIZIO CONVERS. A/D
00000007:                      ADENBIT  EQU      $07             CA1 A LINEA FINE CONV. A/D
                               ;
                                        ORG      DATA

00006000:                      BYTE     DS.B     1               VALORE DA INVIARE AL CONV. A/D
                               ;
                                        ORG      PROGRAM

00004000: 207C 0003
00004004: FF40                 PGM13_8  MOVEA.L  #PIA,A0         PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004                     CLR.B    PIACA(A0)       INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                          MOVE.B   #A_DATDIR,PIADDA(A0)
00004010: 117C 0036
00004014: 0004                          MOVE.B   #A_CNTRL,PIACA(A0)
                               ;
00004016: 08E8 0003
0000401A: 0004                          BSET     #ADSTABIT,PIACA(A0) INIZIO CONV. ALTA
0000401C: 08A8 0003
00004020: 0004                          BCLR     #ADSTABIT,PIACA(A0) INIZIO CONV. BASSA
00004022: 0828 0007
00004026: 0004                 WAIT     BTST     #ADENBIT,PIACA(A0)  CONVERSIONE COMPLETA?
00004028: 67F8                          BEQ      WAIT                SE NO, ATTENDI
                               ;
0000402A: 11E8 0000
0000402E: 6000                          MOVE.B   PIADA(A0),BYTE      ...ALTRIMENTI SALVA VAL. A/D
                               ;
00004030: 4E75                          RTS

                                        END      PGM13_8
```

In questa applicazione ci troviamo di fronte ad un problema che ritroveremo anche in altre occasioni: la possibilità che impulsi spuri, prodottisi durante la fase di inizializzazione del sistema, possano dar luogo a processi indesiderati.

Ecco un esempio. All'accensione del sistema tutto si trova in uno "stato indeterminato"; cioè, i registri contengono dei valori casuali, i convertitori A/D possono dare inizio spontaneamente a processi di conversione, i LED possono essere accesi o spenti. Bisogna, quindi, eseguire il reset del sistema ed inizializzarlo per riportarlo in uno "stato conosciuto". Ma anche durante la stessa fase di inizializzazione si producono degli impulsi o delle variazioni nei livelli dei segnali che possono, ad esempio, dare inizio ad una conversione A/D. Ora, per accertarsi che tutto quanto si trovi nel cosiddetto "stato conosciuto", è necessaria una breve attesa e, in certi casi, delle "false" operazioni di lettura per azzerare i bit di stato e così via.

Questo richiede una conoscenza dettagliata dell'hardware e rende molto più complessa la fase di inizializzazione. Ma, con un piccolo sforzo in più abbiamo la certezza che il sistema operi correttamente in ogni occasione.

Nel nostro esempio con il convertitore A/D possiamo inserire un loop come quello seguente dopo l'istruzione MOVE.B #A_CNTRL, PIACA(A0) per essere sicuri che il convertitore non esegua una conversione una volta terminata la fase di inizializzazione:

```
MOVEQ   #MAXCONV,D0 ATTENDERE PER ESSERE SICURO CHE
JSR     DELAY       ..UN'EVENTUALE CONVERSIONE È FINITA
TST.B   PIADA(A0)   FINTA LETTURA: AZZERA BIT DI STATO IN
                    CTR
```

Il simbolo MAXCONV dovrebbe essere uguale al numero massimo di millisecondi necessari al processo di conversione. La routine di ritardo utilizzata è quella descritta nel Capitolo 12.

Perchè è necessario attendere? Non sarebbe sufficiente controllare lo stato del bit 7 del registro di controllo? No, perchè il bit 7 non riflette la presenza di un livello, ma piuttosto un cambiamento di livello su CA1. Viene, cioè, messo a 1 da una transizione basso-alto quando la conversione è completa e non dal livello basso in uscita dal convertitore durante il processo di conversione. Quali cambiamenti sarebbero stati necessari se avessimo impiegato, invece, un ingresso sensibile al livello, come CA2?

13-9. Una telescrivente (TTY)

Trasferiremo dei dati da e verso una normale telescrivente seriale da 10 caratteri al secondo.

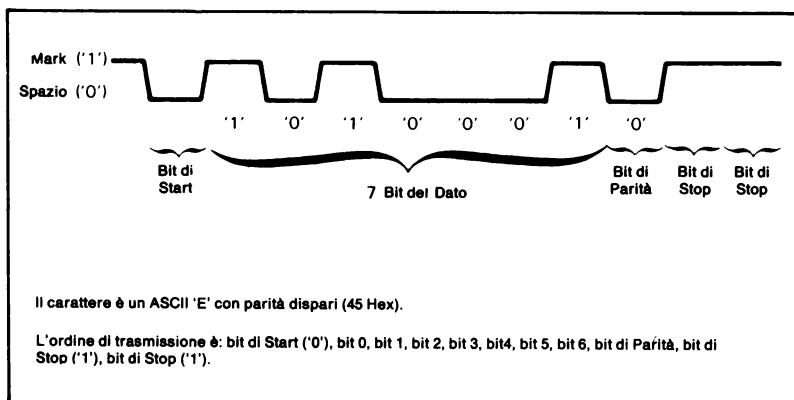
Formato di un Carattere

Una comune telescrivente trasferisce dati in modo seriale asincrono. La procedura è la seguente:

Procedura di trasferimento dati

1. La linea è normalmente allo stato logico '1' (Mark).
2. Un bit di Start (spazio o '0' logico).
3. Si tratta, di solito, di un carattere ASCII a 7 bit, con il bit meno significativo trasmesso per primo.
4. Il bit più significativo è il bit di parità, che può essere pari, dispari o fissato a zero oppure a uno.
5. Due bit di stop ('1' logici) seguono ogni carattere per fornire un minimo di separazione fra l'uno e l'altro.

La Figura 13-19 mostra il formato. Ogni carattere richiede la trasmissione di undici bit, dei quali solo sette si riferiscono al dato

Figura 13-19. Formato di un Dato per Telescrivente.

vero e proprio. Dal momento che la velocità dei dati è di dieci caratteri al secondo, la velocità dei bit è di 11 x 10, cioè 110 Baud. Perciò, ogni bit ha una durata di 1/110 di secondo, o 9,1 millisecondi. Si tratta, comunque, della durata media dato che, normalmente, una telescrivente non garantisce un elevato livello di precisione.

Modo di Ricezione di una TTY

Questa è la procedura di ricezione, il cui diagramma di flusso compare nella Figura 13-20:

Procedura di ricezione dati

Fase 1. Rilevare la presenza di un bit di Start (uno zero logico) sulla linea dati.

Fase 2. Centrare la ricezione, aspettando la durata di mezzo bit, cioè 4,55 millisecondi.

Fase 3. Prelevare i bit del dato, attendendo il tempo di un bit prima di quello successivo. Assemblare i bit del dato in una word, facendo passare, prima, un bit nel Carry mediante uno shift e, quindi, effettuando uno shift circolare del dato con il Carry. Non bisogna dimenticare che il primo ad essere ricevuto è il bit meno significativo.

Fase 4. Calcolare la parità del dato ricevuto e confrontarla con quella trasmessa. Se non coincidono, indicare un "errore di parità".

Fase 5. Prelevare i bit di Stop (attendendo il tempo di un bit fra i due ingressi). Se non sono corretti (se entrambi i bit di Stop non sono uno), indicare un "errore di framing".

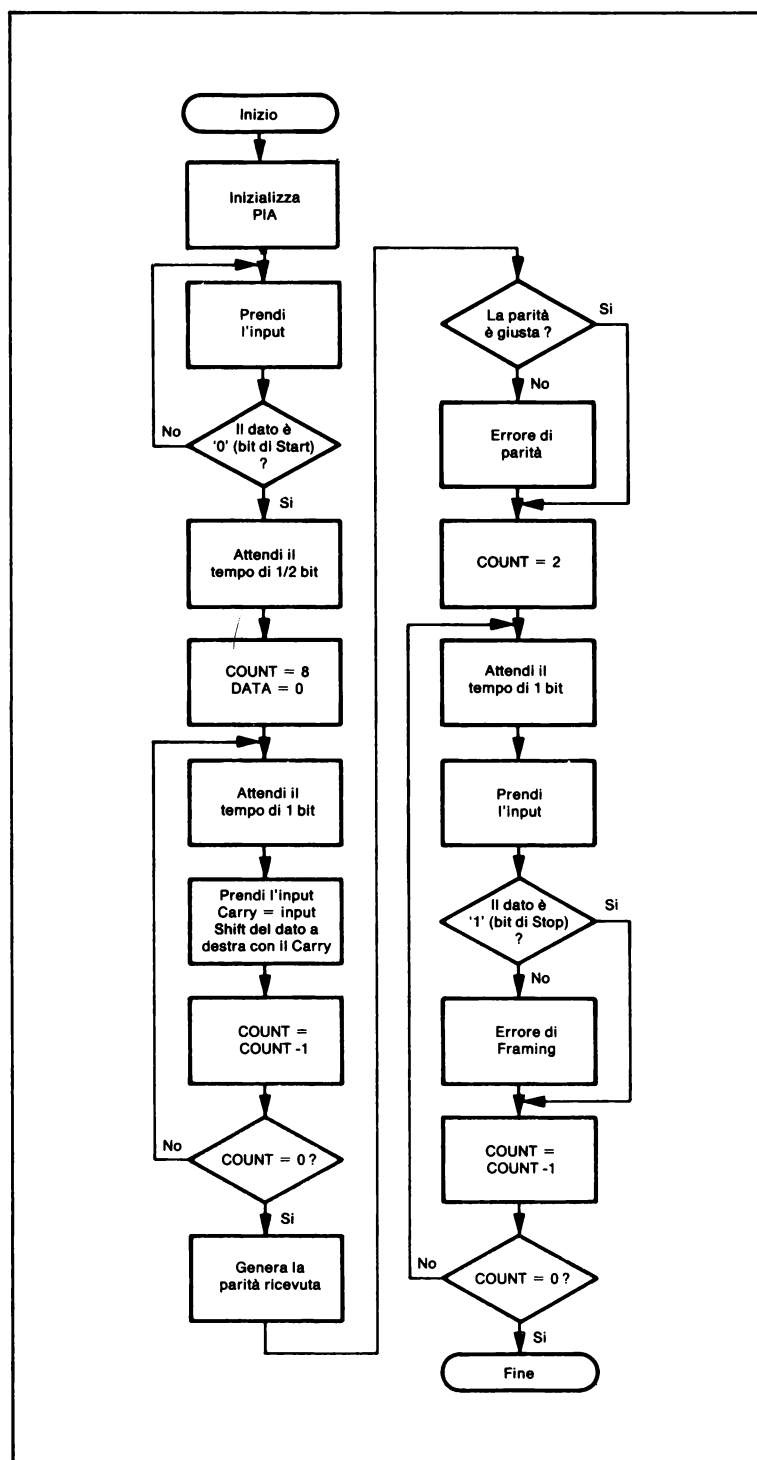



Figura 13-20. Diagramma della Procedura di Ricezione.

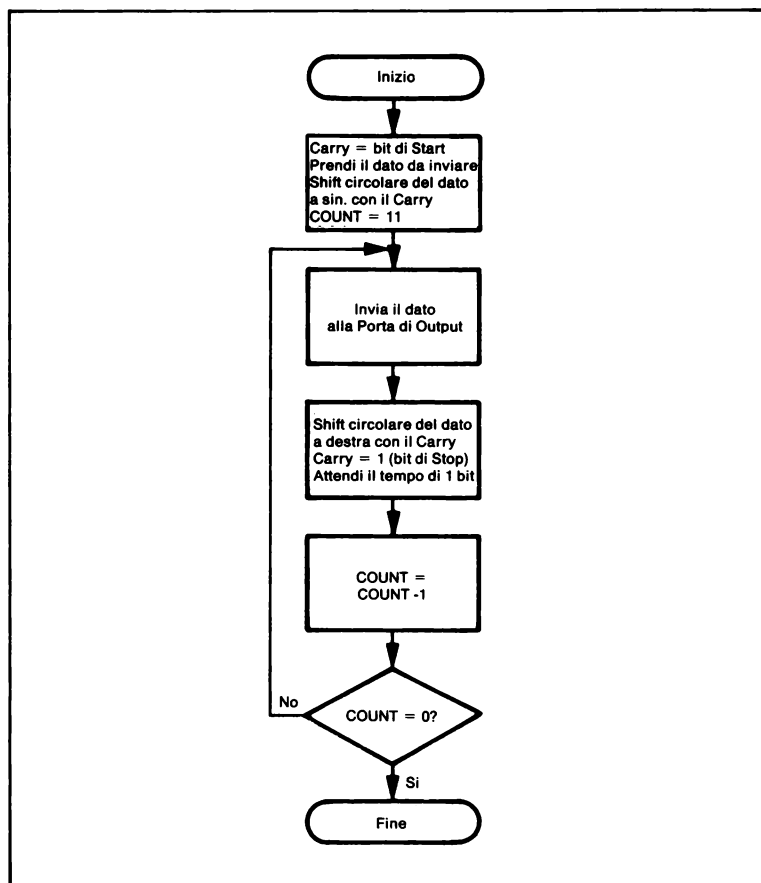



Figura 13-21. Diagramma della Procedura di Trasmissione.

Modo di Trasmissione di una TTY

Questa è la procedura di trasmissione, il cui diagramma di flusso compare nella Figura 13-21:

Procedura di trasmissione

Fase 1. Trasmettere un bit di Start (cioè, un zero logico).

Fase 2. Trasmettere i sette bit del dato, a cominciare dal bit meno significativo.

Fase 3. Generare e trasmettere il bit di parità.

Fase 4. Trasmettere due bit di Stop (cioè, uno logici).

La routine di trasmissione deve attendere la durata di un bit fra due operazioni di output successive.

Funzione 13-9a. Leggere un Dato da una TTY.

Scopo: Prelevare un dato da una telescrivente, usando il bit 7 della porta dati del PIA e mettendo il dato nella variabile CHAR, alla locazione di memoria 6000. La Figura 13-20 descrive la procedura necessaria.

Si presuppone che la porta seriale sia il bit 7 del PIA e che non siano necessari controlli di parità e di framing.

Programma 13-9a

```
00004000:                PROGRAM  EQU      $4000
00006000:                DATA     EQU      $6000
                         ;
0003FF40:                PIA      EQU      $3FF40       INDIRIZZO DI BASE DEL PIA
                         ;
00000000:                PIADDA   EQU      $0           OFFSET REG. DIREZIONE DATI A
00000000:                PIADA    EQU      $0           OFFSET REG. DATI A
00000004:                PIACA    EQU      $4           OFFSET REG. DI CONTROLLO A
                         ;
00000000:                A_DATDIR EQU      $00          DAL LATO A SONO TUTTI INGRESSI
00000004:                A_CNTRL  EQU      $04          SELEZIONA REGISTRO DATI
                         ;
00000007:                TTYBIT   EQU      $7           TTY COLLEGATA AL BIT 7
                         ;
00001C70:                DCNT1    EQU      7280         CONT. RITARDO PER 9.1 MS
00000E38:                DCNT2    EQU      3640         CONT. RITARDO PER 4.55 MS
                         ;
                                  ORG      DATA

00006000:                CHAR     DS.B     1            INPUT DALLA TELESCRIVENTE
                         ;
                                  ORG      PROGRAM
00004000: 207C 0003
00004004: FF40           PGM13_9A MOVEA.L  #PIA,A0      PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004               CLR.B    PIACA(A0)    INIZIALIZZA IL LATO A
0000400A: 117C 0000
0000400E: 0000                    MOVE.B   #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                    MOVE.B   #A_CNTRL,PIACA(A0)
                         ;
00004016: 0828 0007
0000401A: 0000           WAIT     BTST     #TTYBIT,PIADA(A0) C'E' UN BIT DI START?
0000401C: 66F8                    BNE      WAIT         SE NO, ATTENDI
0000401E: 611C                    BSR.S    DLY4_55      ..ALTRIMENTI ATTENDI IL
                         ;                              TEMPO DI MEZZO BIT
00004020: 103C 0080               MOVE.B   #$80,D0      INIZIALIZZA CONT. BIT
                         ;
00004024: 6110           TTYIN    BSR.S    DLY9_1       ATTENDI IL TEMPO DI UN BIT
00004026: 1228 0000               MOVE.B   PIADA(A0),D1 PRENDI IL BIT SUCCESSIVO
0000402A: E311                    ROXL.B   #1,D1        FLAG EXTEND = VALORE DEL BIT
0000402C: E210                    ROXR.B   #1,D0        COMBINA COL DATO PRECEDENTE
0000402E: 64F4                    BCC      TTYIN        CONTINUA FINO A CONT. BIT
00004030: 11C0 6000               MOVE.B   D0,CHAR      SALVA CARATTERE DI INPUT
                         ;
00004034: 4E75                    RTS
                         ;
                         ;   *    ROUTINE DI RITARDO
                         ;
00004036: 323C 1C70      DLY9_1   MOVE.W   #DCNT1,D1    RITARDO DI 9.1 MS
0000403A: 6004                    BRA.S    DELAY
0000403C: 323C 0E38      DLY4_55  MOVE.W   #DCNT2,D1    RITARDO DI 4.55 MS
00004040: 51C9 FFFE      DELAY    DBRA     D1,DELAY
00004044: 4E75                    RTS
                                  END      PGM13_9A
```

Non dimenticate che per primo ad essere ricevuto è il bit 0 del dato.

Le costanti per la routine di ritardo sono state ottenute col metodo descritto nella prima parte di questo capitolo, basandoci su una frequenza di clock pari ad 8 MHz e senza ulteriori stati di attesa. Potete controllare voi stessi. Non è necessario che la durata dei ritardi sia estremamente esatta, poichè la routine centra la ricezione, ogni carattere viene trasmesso separatamente, la velocità di trasferimento dei bit è bassa e, infine, la telescrivente stessa non è molto precisa.

In questo programma ci siamo serviti delle istruzioni ROXL e ROXR. Se siete abituati a programmare altri processori della Motorola, come il 6800 ed il 6809, probabilmente avrete avuto un attimo di perplessità. Ma, nel caso dell'MC68000, nessuna delle istruzioni di shift o di rotazione è in grado di trasferire il bit di Carry in un registro: solo il bit di Extend (X) del registro di stato può essere shiftato in un registro. Ricordate, però, che molte istruzioni dell'MC68000 possono eseguire lo shift in senso inverso, cioè da un registro verso il bit di Carry e quello di Extend.

Quali modifiche sono necessarie perchè il programma controlli anche la parità?

Funzione 13-9b. Inviare un Dato ad una TTY

Scopo: Trasmettere un dato ad una telescrivente usando il bit 0 della porta dati del PIA. Il dato è rappresentato dalla variabile CHAR, alla locazione di memoria 6000. Si presuppone che non sia necessario generare la parità.

Programma 13-9b

```
00004000:                PROGRAM  EQU      $4000
00006000:                DATA     EQU      $6000
                         ;
0003FF40:                PIA      EQU      $3FF40          INDIRIZZO DI BASE DEL PIA
                         ;
00000000:                PIADDA   EQU      $0              OFFSET REG. DIREZIONE DATI A
00000000:                PIADA    EQU      $0              OFFSET REG. DATI A
00000004:                PIACA    EQU      $4              OFFSET REG. DI CONTROLLO A
                         ;
00000080:                A_DATDIR EQU      $80             LA LINEA 7 E' UN'USCITA
00000004:                A_CNTRL  EQU      $04             SELEZIONA REGISTRO DATI
                         ;
00000007:                TTYBIT   EQU      $7              TTY COLLEGATA AL BIT 7
                         ;
00001C70:                DCNT1    EQU      7280            CONT. RITARDO PER 9.1 MS
                         ;
00000008:                CHRBIT   EQU      $08             NUM. DI BIT NEL CARATTERE
00000002:                STPBIT   EQU      $02             NUM. BIT DI STOP DA TRASMETTERE
                         ;
                                  ORG      DATA

00006000:                CHAR     DS.B     1               OUPUT ALLA TELESCRIVENTE
                         ;
                                  ORG      PROGRAM

00004000: 207C 0003
00004004: FF40           PGM13_9B MOVEA.L  #PIA,A0         PRENDI IND. DI BASE DEL PIA
00004006: 4228 0004               CLR.B    PIACA(A0)       INIZIALIZZA IL LATO A
0000400A: 117C 0080
0000400E: 0000                    MOVE.B   #A_DATDIR,PIADDA(A0)
00004010: 117C 0004
00004014: 0004                    MOVE.B   #A_CNTRL,PIACA(A0)
                         ;
00004016: 70FF                    MOVEQ    #-1,D0          FORMA BIT DI STOP
00004018: 1038 6000               MOVE.B   CHAR,D0         PRENDI IL DATO DA INVIARE
0000401C: 740A                    MOVEQ    #1+CHRBIT+STPBIT-1,D2  CONT. BIT CORRETTO PER DBRA
0000401E: 08A8 0007
00004022: 0000                    BCLR     #TTYBIT,PIADA(A0)      INVIA BIT DI START
                         ;
00004024: 6100 001A      SNDBIT   BSR      DLY9_1          ATTENDI IL TEMPO DI 1 BIT
00004028: E258                    ROR.W    #1,D0           CARRY = BIT SUCCESSIVO
0000402A: 6508                    BCS.S    SNDONE          SE DATO = 1 ALLORA INVIA UN UNO
                         ;
0000402C: 08A8 0007
00004030: 0000                    BCLR     #TTYBIT,PIADA(A0)  INVIA '0' COME BIT DATO
00004032: 6006                    BRA.S    NEXT
00004034: 08E8 0007
00004038: 0000           SNDONE   BSET     #TTYBIT,PIADA(A0)  INVIA '1' COME DATO
0000403A: 51CA FFE8      NEXT     DBRA     D2,SNDBIT       CONTINUA FINCHE' NON HAI INVIATO
                         ;                                 TUTTI I BIT
0000403E: 4E75                    RTS
                         ;
                         ;  *     ROUTINE DI RITARDO
                         ;
00004040: 323C 1C70      DLY9_1   MOVE.W   #DCNT1,D1       RITARDO DI 9.1 MS
00004044: 51C9 FFFE      DELAY    DBRA     D1,DELAY
00004048: 4E75                    RTS
                                  END      PGM13_9B
```

Si ricordi che per primo deve essere trasmesso il bit 0.

Nelle applicazioni reali bisogna mettere un '1' logico sulla linea della telescrivente durante la routine di startup, dato che, normalmente, questa linea deve trovarsi allo stato logico 1. Ogni carattere è formato da 11 bit, cominciando con un bit di Start ('0') e terminando con due bit di Stop ('1').

Quali modifiche sarebbe necessario apportare ai programmi precedenti volendo utilizzare come uscite CA2 o CB2?

UART

Queste procedure sono così diffuse e complesse da meritare uno speciale dispositivo LSI: l'UART o Universal Asynchronous Receiver/Trasmitter.[20] L'UART provvede alla procedura di ricezione e, oltre ad un dato in forma parallela, fornisce anche un segnale DATA READY. Accetta anche dati in forma parallela, esegue la procedura di trasmissione e fornisce un segnale PERIPHERAL READY quando è pronto per gestire un altro dato. Gli UART hanno anche altre caratteristiche, fra le quali:

1. La capacità di gestire dati di lunghezza variabile (di solito da 5 a 8 bit), di riconoscere varie opzioni di parità e un numero diverso di bit di Stop (in genere 1, 1-1/2 e 2).
2. Indicatori per errori di framing, errori di parità ed "errori di overrun" (mancata lettura di un carattere prima dell'arrivo di quello successivo).
3. Compatibilità con l'RS-232; cioè, la capacità di generare un segnale di uscita RTS (Request-To-Send), che indica la presenza di un dato al dispositivo di comunicazione e di riconoscere un segnale di input CTS (Clear-To-Send), che indica, in risposta a RTS, la disponibilità del dispositivo stesso. In certi casi, sono possibili anche altri segnali RS-232, quali Received Signal Quality, Data Set Ready o Data Terminal Ready.
4. Uscite tri-state e compatibilità di controllo con un microprocessore.
5. Opzioni di clock che consentono all'UART di campionare più volte i dati in arrivo, per rilevare falsi bit di Start o eventuali altri errori.
6. Possibilità di interrupt e controlli.

Gli UART operano come quattro porte parallele: una porta dati di input, una porta dati di output, una porta di stato ed una porta di controllo. I bit di stato comprendono, oltre a dei flag di Ready, anche degli indicatori d'errore. I bit di controllo selezionano varie opzioni. **Gli UART sono poco costosi (dalle dieci alle centomila lire, a seconda delle caratteristiche) ed il loro impiego estremamente semplice.**

CONSIDERAZIONI FINALI SULL'I/O

Dopo aver visto alcuni esempi di possibili routine di I/O, ci vogliamo soffermare su alcuni aspetti importanti.

I programmi di I/O tendono ad essere piuttosto intricati e sono soggetti a parecchie modifiche durante la fase di collaudo, perciò, è indispensabile l'impiego dei simboli, al posto dei valori reali. Simboli come NUM_DI_RIGA, DUR_RITARDO e COLL_A_MASSA ci forniscono un numero di informazioni molto maggiore rispetto ai corrispondenti valori 13, $07 e $03. Conviene sempre indicare uno stesso valore con simboli diversi, se diverso è il tipo d'impiego; usando, ad esempio, i simboli RIGHE e COLONNE per una tastiera 3 x 3. Se utilizzate un simbolo con un nome non appropriato ed al quale, casualmente, è associato il valore di cui avete bisogno in quel momento, rischiate di fare confusione la prossima volta che dovrete apportare una modifica al programma. Non va dimenticato che l'uso di molti simboli non comporta nessuna conseguenza, tranne quella di aumentare il tempo necessario all'assemblaggio del programma.

È opportuno accertarsi dello stato dei dispositivi di I/O prima di iniziare un qualunque tipo di elaborazione. I dispositivi di I/O, come i convertitori di dati e, in certi casi, anche altri microcomputer collegati al vostro sistema, hanno bisogno di un certo tempo per completare una funzione, iniziata accidentalmente nella fase di inizializzazione.

L'istruzione RESET va usata con cautela:, perchè agisce su tutti i dispositivi collegati alla linea di reset dell'MC68000, non solo su quello che vi interessa.

Se disponete di un monitor di sistema (di cui parleremo più ampiamente nel Capitolo 19), che consente la stampa del contenuto della memoria, bisogna fare molta attenzione in caso di accesso al PIA. Per visualizzare il contenuto di un registro dati del PIA, il monitor deve leggere questo registro, azzerando, automaticamente, i bit di stato 6 e 7 del registro di controllo, creando spesso della confusione.

PROBLEMI

13-1. Un pulsante acceso-spento

Scopo: Ogni pressione del pulsante complementa (inverte) tutti i bit della variabile SWITCH, alla locazione di memoria 6000, che inzialmente contiene uno zero. Il programma deve esaminare continuamente il pulsante e complementare il contenuto di SWITCH ad ogni chiusura. Si può, eventualmente, complementare un'uscita su display, per una migliore visualizzazione del risultato.

Caso Campione:

SWITCH inizialmente contiene il valore zero.

La prima chiusura del pulsante cambia SWITCH in FF_{16}, la seconda lo riporta a zero, la terza di nuovo ad FF_{16}, ecc. Si presuppone che all'eliminazione dei rimbalzi del pulsante provveda l'hardware. Come si potrebbero eliminare dall'interno del programma?

13-2. Debouncing di un interruttore via software

Scopo: Eliminare i rimbalzi (debouncing) di un interruttore meccanico, attendendo fino a che due letture, separate da un opportuno intervallo, non danno lo stesso risultato. Il tempo di debouncing deve trovarsi nella variabile TIME, alla locazione di memoria 6000. Salvare la posizione dell'interruttore nella variabile SWITCH, alla locazione di memoria 6002.

Problema Campione:

SWITCH = 0003 il programma attende 3 ms. prima di ogni lettura.

13-3. Controllo di un interruttore a rotazione

Scopo: Un altro interruttore serve ad attivare un interruttore a rotazione con quattro posizioni, non codificato. La CPU attende che l'interruttore di carico sia chiuso (deve essere zero) e, quindi, legge la posizione di quello a rotazione. Questa procedura consente all'operatore di spostare l'interruttore a rotazione nella sua posizione finale, prima che la CPU tenti di leggerlo. Il programma deve salvare la posizione letta nella variabile SWITCH, alla locazione di memoria 6000. Eliminare i rimbalzi dell'interruttore di attivazione via software.

Problema Campione:

Mettere l'interruttore a rotazione nella posizione 2. Chiudere l'interruttore di attivazione.

Risultato: SWITCH - (6000) = 02

13-4. Indicare con dei LED le posizioni di una serie di interruttori

Scopo: Un gruppo di otto LED indica la posizione di altrettanti interruttori. Se l'interruttore è chiuso (zero), il LED deve essere acceso; altrimenti,deve essere spento. Si presuppone che la porta di output della CPU sia collegata ai catodi dei LED.

Problema Campione:

INTERRUTTORE 0 CHIUSO	Risult.	LED 0 ACCESO
INTERRUTTORE 1 APERTO		LED 1 SPENTO
INTERRUTTORE 2 CHIUSO		LED 2 ACCESO
INTERRUTTORE 3 APERTO		LED 3 SPENTO
INTERRUTTORE 4 CHIUSO		LED 4 SPENTO
INTERRUTTORE 5 APERTO		LED 5 ACCESO
INTERRUTTORE 6 APERTO		LED 6 ACCESO
INTERRUTTORE 7 APERTO		LED 7 SPENTO

Quali modifiche si dovrebbero apportare al programma per fare in modo che un interruttore, collegato al bit 7 della porta A del PIA, determini se i display sono attivi (se, cioè, l'interruttore di controllo è chiuso i display collegati alla Porta B riflettono lo stato degli interruttori collegati alla Porta A; se l'interruttore di controllo è aperto i display sono sempre spenti)? Un interruttore di controllo è utile quando i display possono distrarre l'operatore, come in un aereo.

Quali sono le modifiche necessarie per trasformare l'interruttore di controllo in un pulsante acceso-spento, per cui ogni chiusura invertirà lo stato precedente dei display? Si presuppone che i display all'inizio si trovino nello stato attivo e che il programma esamini il pulsante, eliminandone i rimbalzi, prima di inviare i dati ai display.

13-5. Conteggio su un display a sette segmenti

Scopo: Il programma deve, continuamente, contare da o a 9 su un display a sette segmenti, iniziando da zero.

Un suggerimento: Provate degli intervalli di diversa durata ed osservate cosa accade. Quando diventano comprensibili le cifre? Cosa accade se il display rimane vuoto per un certo tempo?

13-6. Separazione delle chiusure di una tastiera non codificata

Scopo: Il programma deve leggere gli ingressi da una tastiera 3 x 3 non codificata e salvarli in un vettore. Il numero di ingressi richiesti si trova nella variabile COUNT, alla locazione di memoria 6000, ed il vettore è definito dalla variabile KEY, alla locazione 6001.

Problema Campione:

```
COUNT – (6000) = 04
Gli ingressi sono 7,2,2,4

Risultato: KEY – (6001) = 07
                 (6002) = 02
                 (6003) = 02
                 (6004) = 04
```

13-7. Leggere una frase da una tastiera codificata

Scopo: Il programma deve leggere gli ingressi da una tastiera ASCII (7 bit con bit di Parità zero) e metterli in un vettore, finchè non riceve il carattere ASCII di punto ($2E). Il vettore è definito dalla variabile KEY, alla locazione di memoria 6001. Ogni ingresso è indicato da uno strobe, come nell'Esempio 13-6.

Problema Campione:

Gli ingressi sono HELLO

```
Risultato: KEY - (6001) = 48 'H'
                 (6002) = 45 'E'
                 (6003) = 4C 'L'
                 (6004) = 4C 'L'
                 (6005) = 4F 'O'
                 (6006) = 2E '.'
```

13-8. Genratore d'onda quadra di ampiezza variabile

Scopo: Il programma deve generare un'onda quadra, come viene mostrato nella figura seguente, usando un convertitore D/A. La variabile SCALE, alla locazione di memoria

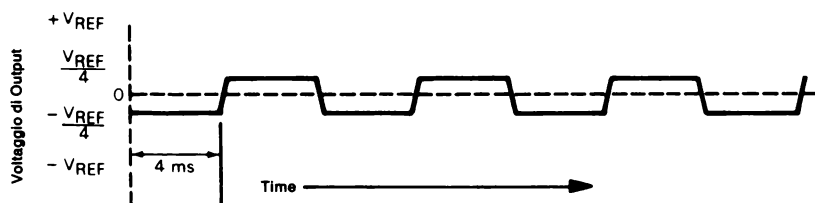
6000, contiene l'ampiezza scalare dell'onda. La variabile LENGTH, alla locazione 6001, contiene la lunghezza di un mezzo ciclo in millisecondi. La variabile CYCLES, alla locazione 6002, contiene il numero dei cicli.

Si presuppone che un'uscita digitale pari ad 8016 diretta al convertitore provochi un output analogico di zero volt. In generale, un output digitale di D provoca un output analogico di (D-80)/80 X - VREF/4 volt.

Problema Campione:

SCALE	– (6000)	$= A0_{16}$
LENGTH	– (6001)	= 04
CYCLES	– (6002)	= 03

(0040)	=	$A0_{16}$
(0041)	=	04
(0042)	=	03

Risultato:

Il voltaggio base è 80_{16} = 0 Volts
Valore massimo: $100_{16} = -V_{REF}$ Volts
Quindi $A0_{16} = (A0-80)/80 \times (-V_{REF}) = -V_{REF}/4$

Il programma genera tre impulsi di ampiezza VREF/4, con una lunghezza di mezzo ciclo pari a quattro millisecondi.

13-9. Media di letture analogiche

Scopo: Il programma deve prelevare quattro valori da un convertitore A/D ad intervalli di 10 millisecondi e mettere la media nella variabile DATA, alla locazione di memoria 6000. Si presuppone che il tempo di conversione A/D sia trascurabile e possa essere ignorato.

Problema Campione:

Le letture esadecimali sono 86, 89, 81, 84

Risultato: DATA - (6000) = 85_{16}

13-10. Un terminale a 30 caratteri al secondo

Scopo: Apportare alle routine di trasmissione e ricezione dell'Esempio 13-9 le modifiche necessarie per gestire un terminale a 30 cps., che trasferisce dati ASCII con un bit di Stop e parità di tipo pari. Come dovrebbero essere queste routine, in modo da gestire l'uno o l'altro di due terminali, a seconda del valore di un flag contenuto nella variabile TTYTYPE, alla locazione di memoria 6000; ad es., TTYTYPE = 0 per il terminale a 30 cps. e TTYTYPE = 1 per il terminale a 10 cps.?

BIBLIOGRAFIA

1. A. Osborne et al. *4 & 8 Bit Microprocessor Handbook,* Berkeley: Osborne/McGraw-Hill, 1981, pp. 9-45 through 9-54.
2. J. Gilmore and R. Huntington. "Designing with the 6820 Peripheral Interface Adapter," *Electronics,* December 23, 1976, pp. 85-86.
3. *The TTL Data Book for Design Engineers,* Texas Instruments, Inc., P.O. Box 5012, Dallas, TX 75222, da p. 7-151 a p. 7-156.
4. E. Dilatush. "Special Report; Numeric and Alphanumeric Displays," *EDN, January 5, 1978, pp. 26-35.*
5. *The TTL Data Book Design Engineers,* Texas Instruments, Inc., P.O. Box 5012, Dallas, TX 75222, da p. 7-22 a p. 7-34.
6. A. Pshaenich. "Interface Considerations for Numeric Display Systems," Motorola Semiconductor Products, Inc., Application Note AN-741, Phoenix, Ariz. 1975.
7. Motorola Semiconductor Products Inc., *Microprocessor Applications Manual,* New York: McGraw-Hill, 1975, da p. 5-6 a p. 5-11.
8. Motorola Semiconductor Products Inc., *Microprocessor Applications Manual,* Nw York: McGraw-Hill, 1975, da p. 5-1 a p. 5-5.
9. Vedi Rif. 2.
10. G. Kane et al. *An Introduction to Microcomputers; Volume 3 - Some Real Support Devices,* Section El.Berkeley: Osborne/McGraw-Hill, 1979.
11. E.R. Hnatek. *A. User's Handbook of D/A and A/D Converters, New York: Wiley, 1976.*
12. *P.H. Garrett. Analog Systems for Microprocessors and Minicomputers,* Reston, Va.: Reston Publishing Co. (Prentice-Hall), 1978.

13. B. Amazeen. "Monolithic D-A Converter Operates on Single Supply," *Electronics,* February 28, 1980, pp. 125-31.
14. Vedi Rif. 11.
15. Vedi Rif. 12.
16. G. Kane et al. *An Introduction to Microcomputers: Volume 3 - Some Real Support Devices,* Section E2. Berkeley: Osborne/Mc-Graw-Hill, 1979.
17. D. Aldridge. "Analog to Digital Conversion Techniques with the M6800 Microprocessor System," Motorola Semiconductor Products, Inc. Application Note AN-757, Phoenix, Ariz., 1975.
18. P. Bradshaw. "Two-Chip A/D Converter," *Electronic Design,* March 29, 1979, pp. 128-36.
19. M. Tuthill and D.P. Burton. "Low-Cost A/D Converter Links Easily with Microprocessor," *Electronics,* August 30, 1979, pp. 149-55.
20 Per una descrizione degli UART vedi P. Rony et al. "The Bugbook IIa," E and L Instrumets Inc., 61 First Street, Derby, CT 06418, or D.G. Larsen et al. "INWAS: Interfacing with Asynchronous Serial Mode," *IEEE Transactions on Industrial Electronics and Control Instrumentation,,* February 1977, pp. 2-12.
21 "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange," EIA RS-232C, Electronic Industries Association, 2001 I Street N.W., Washington, D.C. 20006, August 1969.

G. Kane et al. *An Introduction to Microcomputers: Volume 3 - Some Real Support Devices,* Berkeley: Osborne/McGraw-Hill, 1978, da p. J5-9 a p. J5-14.
G. Pickles. "Who Afraid of RS-232?", *Kilobaud,* May 1977, pp. 50-54.
C.A. ogdin. "Microcomputer Buses - Part II," *Mini-Micro Systems,* July 1978, pp. 76-80.

CAPITOLO 14

L'USO DELL'ACIA 6850

L'ACIA **6850,** o Asynchronous Communications Interface A-dapter (cfr. Figura 14-1), **è un UART progettato appositamente per essere impiegato con i microcomputer che utilizzano l'MC68000, il 6800, il 6809 ed il 6852. Occupa due indirizzi di memoria e contiene due registri di sola lettura (dato ricevuto e stato) e due registri di sola scrittura (dato trasmesso e controllo).** Le Tabelle 14-1 e 14-2 descrivono i contenuti di questi registri.

INDIRIZZAMENTO DELL'ACIA 6850

Specificazione degli indirizzi

I registri interni dell'ACIA sono indirizzati mediante le linee RS (Register Select) e R/W (Read/Write), come è indicato nella Tabella 14-3. Se, come di solito avviene, RS è collegata al bit meno significativo del bus indirizzi dell'MC68000, allora l'indirizzo dei registri dati è maggiore di uno rispetto a quello dei registri di controllo e di stato. In seguito all'impiego della linea R/W per l'indirizzamento i cicli di lettura e di scrittura accedono a registri differenti, per cui un programma non può leggere i registri di dato trasmesso o di controllo, nè scrivere nel registro di dato ricevuto o in quello di stato. Qualora il programma debba richiamare il contenuto dei registri di sola scrittura deve conservarne una copia nella RAM. **Indicheremo i vari indirizzi con ACIADR (registro di dato ricevuto in fase di lettura, e registro di dato trasmesso in fase di scrittura), ACIASR (il registro di stato, di sola lettura) e ACIACR (il registro di controllo, di sola scrittura). ACIASR ed ACIACR corrispondono allo stesso indirizzo fisico.**

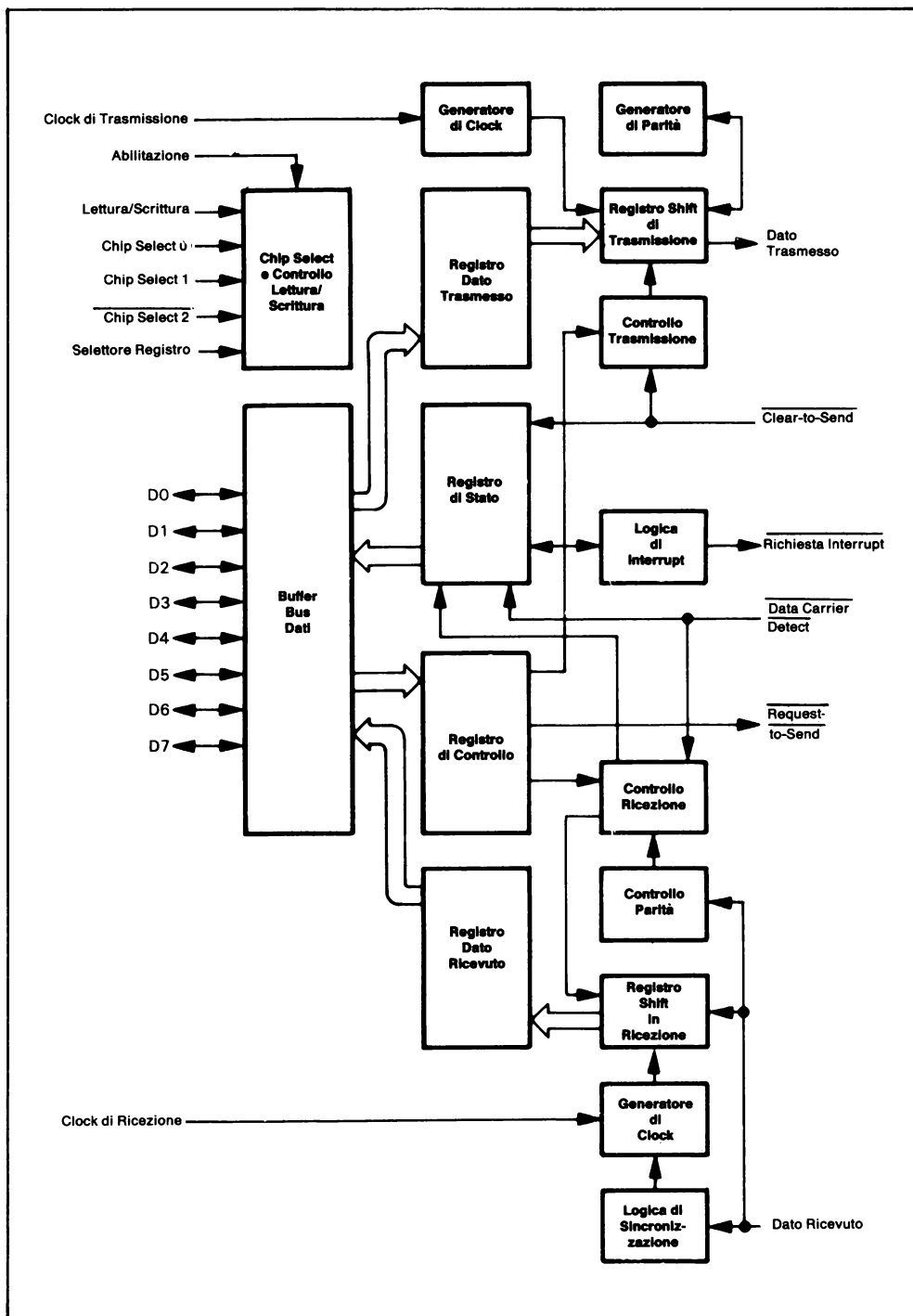



Figura 14-1. Diagramma a Blocchi dell'ACIA 6850

Tabella 14.1. Definizione dei Contenuti dei Registri dell'ACIA 6850.

Numero Linee Bus Dati	Indirizzo Buffer			
	RS·$\overline{R/W}$ Registro Dato Trasmesso	RS·R/W Registro Dato Ricevuto	$\overline{RS}$·$\overline{R/W}$ Registro di Controllo	$\overline{RS}$·R/W Registro di Stato
	(Solo Scrittura)	(Solo Lettura)	(Solo Scrittura)	(Solo lettura)
0	Bit 0*	Bit 0	Selezione Counter Divide 1 (CRO)	Registro Ricezione Dati Pieno (RDRF)
1	Bit 1	Bit 1	Selezione Counter Divide 2 (CR1)	Registro Trasmissione Dati Vuoto (TDRE)
2	Bit 2	Bit 2	Selezione Word 1 (CR2)	Data Carrier Detect (DCD)
3	Bit 3	Bit 3	Selezione Word 2 (CR3)	Clear-to-Send (CTS)
4	Bit 4	Bit 4	Selezione Word 3 (CR4)	Errore di Framing (FE)
5	Bit 5	Bit 5	Controllo Trasmissione 1 (CR5)	Overrun del Ricevitore (OVRN)
6	Bit 6	Bit 6	Controllo Trasmissione 2 (CR6)	Errore di Parità (PE)
7	Bit 7***	Bit 7**	Abilitazione Interrupt Ricezione (CR7)	Richiesta di Interrupt (IRQ)

* Bit di testa = LSB = Bit 0
** Questo bit sarà "zero" nella parità "7-bit plus"
*** Questo bit avrà un valore qualsiasi nella parità "7-bit plus"

CARATTERISTICHE SPECIALI

L'ACIA 6850 ha le seguenti particolarità:

1. **I cicli di lettura e scrittura indirizzano registri fisicamente distinti.** Perciò non si possono utilizzare i registri dell'ACIA come indirizzi per delle istruzioni che eseguono operazioni da memoria a memoria, come l'OR di un dato contenuto in una locazione.
2. **Il registro di controllo dell'ACIA non può essere letto dalla CPU.** Se il programma ha necessità di questo valore bisognerà salvare una copia del registro di controllo in memoria.

Reset dell'ACIA

3. **L'ACIA non dispone di un ingresso di reset. Il reset può essere effettuato solamente ponendo degli uno nei bit 0 ed 1 del registro di controllo.** Questa procedura (chiamata Master Reset) deve essere necessariamente eseguita prima di usare l'ACIA, allo scopo di evitare la presenza di un carattere iniziale del tutto casuale.
4. **I segnali RS-232 sono tutti attivi bassi.** In particolare, RTS (Request-To-Send) deve essere messo alto per renderlo inattivo, se non viene utilizzato.

Tabella 14-2. Significato dei Bit di Registro di Controllo dell'ACIA 6850.

CR6	CR5	Funzione
0	0	RTS = basso. Interrupt di Trasmissione Disabilitato
0	1	RTS = basso. Interrupt di Trasmissione Abilitato
1	0	RTS = alto. Interrupt di Trasmissione Disabilitato
1	1	RTS = basso. Trasmetti un livello di Break sull'uscita Dato Trasmesso. Interrupt di Trasmissione Disabilitato

CR4	CR3	CR2	Funzione
0	0	0	7 Bit + Parità Pari + 2Bit di Stop
0	0	1	7 Bit + Parità Dispari + 2 Bid di Stop
0	1	0	7 Bit + Parità Pari + 1 Bit di Stop
0	1	1	7 Bit + Parità Dispari + 1 Bit di Stop
1	0	0	8 Bit + 2 Bit di Stop
1	0	1	8 Bit + 1 Bit di Stop
1	1	0	8 Bit + Parità Pari + 1 Bit di Stop
1	1	1	8 Bit + Parità Dispari + 1 Bit di Stop

CR1	CR0	Funzione
0	0	÷1
0	1	÷16
1	0	÷64
1	1	Master Reset

Tabella 14-3. Indirizzamento interno per l'ACIA 6850.

RS (selezione Registro)	R/W Lettura/Scrittura 1 = Lettura, 0 = Scrittura	Registro Indirizzato	Offset Indicizzato rispetto al Registro di Controllo dell'ACIA
0	0	Registro di Controllo (solo Scrittura)	0
0	1	Registro di Stato (solo Lettura)	0
1	0	Registro Dato Trasmesso (sola Scrittura)	1
1	1	Registro Dato Ricevuto (solo Lettura)	1

5. **L'ACIA richiede un clock esterno.** In genere, la frequenza è di 4800Hz e viene usato il modo 16 (il bit 1 del registro di controllo è uguale a 0, il bit 0 è uguale a 1). L'ACIA si serve del clock per centrare la ricezione a metà dell'intervallo di trasmissione di un bit e per evitare dei falsi bit di Start, provocati dalla presenza di rumore sulle linee.

Posizione dei flag di disponibilità nel registro di stato

6. **Il Flag che indica la disponibilità di un dato (il registro dato ricevuto è pieno, o RDRF, Received Data Register Full) è il bit 0 del registro di stato. Il flag che indica la disponibilità della periferica (il registro dato trasmesso è vuoto, o TDRE, Transmitted Data Register Empty) è il bit 1 del registro di stato.**

14-1. Ricevere un dato da una TTY

Scopo: Ricevere un dato da una telescrivente usando un ACIA 6850 e salvare il dato nella variabile stringa STRING, che inizia alla locazione di memoria 6000. Terminare l'input dopo aver ricevuto un carattere di ritorno carrello.

Programma 14-1

```
00004000:                PROGRAM  EQU      $4000
00006000:                DATA     EQU      $6000
                         ;
0003FF01:                ACIA     EQU      $3FF01           INDIRIZZO DI BASE DELL'ACIA
00000000:                ACIACR   EQU      $0               REGISTRO DI CONTROLLO DELL'ACIA
00000000:                ACIASR   EQU      $0               REGISTRO DI STATO DELL'ACIA
00000002:                ACIADR   EQU      $2               REGISTRO DATI DELL'ACIA
00000045:                AMODE    EQU      $45              MODO OPERATIVO DELL'ACIA
00000003:                MRESET   EQU      $03              MASTER RESET DELL'ACIA
0000000D:                CR       EQU      $0D              COD. ASCII DI RITORNO CARRELLO
                         ;
                                  ORG      DATA

00006000:                STRING   DS.B     80               STRINGA PER LA MEMORIZZAZIONE
                         ;
                                  ORG      PROGRAM

00004000: 207C 0003
00004004: FF01           PGM14_1  MOVEA.L  #ACIA,A0         INDIRIZZO DELL'ACIA
00004006: 117C 0003
0000400A: 0000                    MOVE.B   #MRESET,ACIACR(A0) MASTER RESET DELL'ACIA
0000400C: 117C 0045
00004010: 0000                    MOVE.B   #AMODE,ACIACR(A0)  MODO OPERATIVO DELL'ACIA
                         ;
00004012: 327C 6000               MOVEA.W  #STRING,A1       INDIRIZZO STRINGA DI INPUT
00004016: 6108           LOOP     BSR.S    INCH             LEGGI CARATTERE
00004018: 0C19 000D               CMPI.B   #CR,(A1)+        E' UN RITORNO CARRELLO?
0000401C: 66F8                    BNE.S    LOOP             NO, CONTINUA L'INPUT
                         ;
0000401E: 4E75                    RTS
                         ;
00004020: 1028 0000      INCH     MOVE.B   ACIASR(A0),D0    LEGGI IL REG. DI STATO DELL'ACIA
00004024: E208                    LSR.B    #1,D0            IL REG. DI RICEZIONE E' VUOTO?
00004026: 64F8                    BCC.S    INCH             NO, CONTINUA IL CONTROLLO
00004028: 12A8 0002               MOVE.B   ACIADR(A0),(A1)  LEGGI CARATTERE
0000402C: 4E75                    RTS

                                  END      PGM14_1
```

Il programma deve effettuare, inizialmente, il reset dell'ACIA mettendo degli uno nelle posizioni 0 ed 1 del registro di controllo. L'ACIA dispone di un reset interno al momento dell'accensione, che mantiene l'ACIA in uno stato non operativo finchè non viene eseguito un Master Reset.

Determinazione del modo operativo

Il programma stabilisce il modo operativo dell'ACIA ponendo ad uno i bit del registro di controllo nel modo seguente:

Bit 7 = 0 per disattivare l'interrupt del ricevitore

Bit 6 = 1, Bit 5 = 0 per rendere alta (inattiva) la linea RTS (Request-To-Send) e disattivare l'interrupt del trasmettitore

Bit 4 = 0 per word da 7 bit

Bit 3 = 0, Bit 2 = 1 per una parità dispari con due bit di Stop
Bit 1 = 0, Bit 0 = 1 per un clock 16 (deve essere fornita una frequenza di 4800 Hz)

Il flag di stato del dato ricevuto è il bit 0 del registro di stato dell'ACIA. Cosa accadrebbe se provassimo a sostituire

```
MOVE.B    ACIASR(A0),D0
LSR.B     #1,D0
```

con l'unica istruzione

```
LSR       ACIASR(A0)
```

Non bisogna dimenticare che i registri di stato e di controllo utilizzano lo stesso indirizzo ma sono fisicamente distinti, e per selezionare l'uno o l'altro viene utilizzata la linea R/W (Read/Write).

Provate ad aggiungere una routine per il rilevamento di un errore. Ponete

(STATUS) = 0 se non si verificano errori

= 1 se si verifica un errore di parità
(il Bit 6 del registro di stato = 1)

= 2 se si verifica un errore di overrun
(il bit 5 del registro di stato = 1)

= 3 se si verifica un errore di framing
(il bit 4 del registro di stato = 1)

Si presume che la priorità degli errori proceda dai bit di ordine alto a quelli di ordine basso del registro di stato dell'ACIA (cioè, gli errori di parità hanno una priorità superiore a quella degli errori di overrun, che, a loro volta, hanno priorità superiore a quelli di framing).

14-2. Invio di un dato ad una TTY

Scopo: Inviare un dato dalla variabile stringa STRING, alla locazione di memoria 6000, ad una telescrivente usando un ACIA 6850. La stringa termina con un carattere di ritorno carrello.

Programma 14-2

```
00006000:            DATA     EQU    $6000
00004000:            PROGRAM  EQU    $4000
                     ;
0003FF01:            ACIA     EQU    $3FF01      INDIRIZZO DI BASE DELL'ACIA
00000000:            ACIACR   EQU    $0          REGISTRO DI CONTROLLO DELL'ACIA
00000000:            ACIASR   EQU    $0          REGISTRO DI STATO DELL'ACIA
00000002:            ACIADR   EQU    $2          REGISTRO DATI DELL'ACIA
00000045:            AMODE    EQU    $45         MODO OPERATIVO DELL'ACIA
00000003:            MRESET   EQU    $03         MASTER RESET DELL'ACIA
00000001:            TDRE     EQU    $1          IL REG. DATO TRASMESSO E' VUOTO
0000000D:            CR       EQU    $0D         COD. ASCII DI RITORNO CARRELLO
                     ;
                              ORG    DATA

00006000:            STRING   DS.B   80          STRINGA PER LA MEMORIZZAZIONE
                     ;
                              ORG    PROGRAM
```

```
00004000: 207C 0003
00004004: FF01          PGM14_2  MOVEA.L #ACIA,A0            INDIRIZZO DELL'ACIA
00004006: 117C 0003
0000400A: 0000                   MOVE.B  #MRESET,ACIACR(A0) MASTER RESET DELL'ACIA
0000400C: 117C 0045
00004010: 0000                   MOVE.B  #AMODE,ACIACR(A0)  MODO OPERATIVO DELL'ACIA
                        ;
00004012: 327C 6000              MOVEA.W #STRING,A1          INDIRIZZO STRINGA DI OUTPUT
00004016: 6108          LOOP     BSR.S   OUTCH               CARATTERE DA INVIARE
00004018: 0C19 000D              CMPI.B  #CR,(A1)+           E' UN RITORNO CARRELLO?
0000401C: 66F8                   BNE.S   LOOP                NO, CONTINUA L'OUTPUT DELLA STRINGA
                        ;
0000401E: 4E75                   RTS
                        ;
00004020: 0828 0001
00004024: 0000          OUTCH    BTST    #TDRE,ACIASR(A0) PRONTO PER L'INVIO?
00004026: 67F8                   BEQ.S   OUTCH               NO, CONTINUA IL CONTROLLO
00004028: 1151 0002              MOVE.B  (A1),ACIADR(A0) INVIA CARATTERE
0000402C: 4E75                   RTS

                                 END     PGM14_2
```

Il flag di stato del trasmettitore è il bit 1 del registro di stato dell'ACIA. L'istruzione BTST in questo caso si rivela particolarmente utile, dal momento che essa è in grado di esaminare un determinato bit senza alterare il contenuto del registro di stato dell'ACIA. In che modo si potrebbe modificare il programma per la ricezione di un dato, utilizzando l'istruzione Bit Test?

BIBLIOGRAFIA

1. A. Osborne and Gerry Kane. *4 & 8-Bit Microprocessor Handbook,* Berkeley: Osborne/McGraw-Hill, 1981, da p. 9-55 a p. 9-61.
2. K. Fronheiser. "Device Operation and System Implementation of the Asynchronous Communications OInterface Adapter," Motorola Semiconductor Products, Inc. Application Note AN-754, Phoenix, AZ, 1975.
3. J. Volp. "Software Switches Baud Rate," *EDN*, November 5, 1979, P. 83.

CAPITOLO 15

INTERRUPT ED ALTRE EXCEPTION

Linee di interrupt

Le linee di interrupt sono ingressi che la CPU esamina durante ogni ciclo di istruzione e che le permettono di reagire ad eventi asincroni più efficacemente di quanto sarebbe possibile mediante il semplice polling delle periferiche. Generalmente, gli interrupt richiedono una struttura hardware più complessa rispetto ai normali I/O (programmati), ma, in compenso, consentono una risposta più rapida e diretta.[1]

Nell'MC68000, gli interrupt rappresentano solo una delle categorie di eventi indicati come *Exception*. Sebbene questa terminologia non venga adottata per altri microprocessori, nel caso dell'MC68000 si rivela particolarmente opportuna in quanto il numero ed il tipo di eventi in grado di provocare una *Exception* vanno ben oltre le normali richieste di interrupt provenienti dalle diverse periferiche. Tuttavia, prima di analizzare le risposte fornite dall'MC68000 ai vari tipi di Exception, ci soffermeremo su alcune delle caratteristiche generali degli interrupt, dal momento che questi rappresentano le Exception più frequenti.

Utilità degli interrupt

A cosa servono gli interrupt? **Gli interrupt consentono a dei particolari eventi (allarmi, una mancanza di corrente, il trascorrere di un certo intervallo di tempo, ecc.) e a delle periferiche, che hanno dei dati disponibili o che sono pronte ad accettare dei dati, di attirare immediatamente l'attenzione della CPU. Non è necessario che un programma esamini ogni potenziale sorgente di dati (polling) nè che il programmatore si preoccupi che al sistema possa sfuggire un qualche evento.**

Un sistema di interrupt è come il campanello di un telefono: suona ogni volta che c'è una chiamata, in modo da non dover alzare continuamente il ricevitore per controllare se c'è qualcuno in linea. La CPU è libera di occuparsi delle normali faccende (ed, in questo modo, riesce a sbrigarne molte di più). Quando accade qualcosa l'interrupt avverte la CPU e la costringe a servire l'input prima di riprendere le normali operazioni. Naturalmente, il quadro diventa ben più complesso (proprio come un centralino telefonico) quando si hanno molti interrupt di importanza diversa e quando ci sono delle funzioni che non possono essere arrestate da un interrupt.

CARATTERISTICHE DEI SISTEMI DI INTERRUPT

Le modalità di realizzazione di un sistema d'interrupt variano notevolmente. Ecco alcuni degli aspetti che caratterizzano un sistema:

1 Quanti ingressi di interrupt ci sono?
2. In che modo la CPU risponde ad un interrupt?
3. Come fa la CPU a stabilire l'origine di un interrupt se il numero delle possibili sorgenti supera quello degli ingressi?
4. La CPU riesce a distinguere fra interrupt importanti e non?
5. Come e quando un sistema di interrupt è abilitato o meno?

Esistono molte risposte diverse a queste domande. Lo scopo, comunque, è sempre quello di fare in modo che la CPU risponda rapidamente agli interrupt e riprenda, poi, la sua normale attività.

Il numero di ingressi di interrupt presenti sul chip del microprocessore determina il numero di differenti risposte che la CPU è in grado di fornire senza la necessità di hardware o software supplementari. Ciascun ingresso produce una risposta interna diversa.

Indipendentemente dal tipo di risposta, la CPU, tutte le volte che arriva un interrupt, trasferisce il controllo alla corrispondente routine di servizio, non senza aver provveduto a salvare il contenuto del contatore di programma. Esegue, in pratica, l'equivalente di un'istruzione Call o Jump-To-Subroutine che abbia come indirizzo quello iniziale della routine di servizio dell'interrupt. L'indirizzo di ritorno viene salvato sullo stack ed il controllo passa alla subroutine indicata. Alcune CPU generano internamente sia l'istruzione che l'indirizzo, altre richiedono, invece, delle componenti hardware esterne. Normalmente, la CPU genera un indirizzo o un'istruzione diversi per ciascuno degli ingressi disponibili.

Polling e Vettori

Se il numero dei dispositivi in grado di provocare un interrupt è superiore al numero degli ingressi, la CPU ha bisogno di hardware o software supplementari per riuscire ad individuarne l'origine. Nel caso più semplice il software è costituito da una routine di polling, che controlla lo stato delle varie periferiche in grado di causare un interrupt. L'unico vantaggio di questo sistema, rispetto ad un normale polling, è che la CPU sa che almeno un dispositivo è attivo. **La soluzione alternativa richiede che dell'hardware addizionale fornisca uno specifico dato d'ingresso (o "vettore") per ciascun dispositivo.** Vengono adottate anche soluzioni miste, con dei vettori che identificano gruppi di ingressi, fra i quali la CPU ne individua uno in particolare mediante polling.

Priorità

Livelli di priorità

Un sistema di interrupt capace di distinguere fra interrupt più o meno importanti viene chiamato "sistema di interrupt a priorità". L'hardware interno fornisce un numero di livelli di priorità pari a quello degli ingressi, mentre dell'hardware esterno consente ulteriori livelli mediante l'impiego di un registro di priorità e di un comparatore. L'hardware esterno non permette ad un interrupt di raggiungere la CPU, a meno che la sua priorità non sia maggiore del valore contenuto nel registro di priorità. Un sistema di interrupt a priorità, in certi casi, gestisce con delle modalità particolari quegli interrupt che hanno un basso livello di priorità e possono essere ignorati per lunghi periodi di tempo.

Abilitare e Disabilitare

Gran parte dei sistemi di interrupt possono essere abilitati o disabilitati. Di solito, le varie CPU disabilitano automaticamente gli interrupt quando viene eseguito un RESET (così la routine di avviamento può inizializzare il sistema) e quando ricevono un interrupt (in modo che un altro successivo non arresti la stessa routine di servizio). Il programmatore, se lo desidera, può disattivare gli interrupt durante la preparazione o l'elaborazione di dati, durante l'esecuzione di un loop di temporizzazione o durante un'operazione multibyte.

Un interrupt che non può essere disabilitato (definito anche "non-maskable", cioè non mascherabile) si rivela utile nel caso di una mancanza di corrente, un evento che deve, naturalmente, avere la precedenza su ogni altra attività.

Svantaggi degli Interrupt

I vantaggi degli interrupt sono abbastanza evidenti, ma esistono anche degli svantaggi:

1. I sistemi di interrupt richiedono spesso una notevole quantità di hardware aggiuntivo.
2. Gli interrupt richiedono trasferimenti di dati, sotto il controllo del programma attraverso la CPU. Non si hanno i vantaggi del DMA, in termini di velocità.
3. Gli interrupt sono degli input casuali, che rendono difficoltoso il debbugging ed il collaudo di un programma. Si possono verificare degli errori del tutto occasionali, difficili, quindi, da localizzare e correggere.[2]
4. Gli interrupt aumentano, in certi casi, la lunghezza del programma, soprattutto quando devono essere salvati molti registri e bisogna individuare l'origine di un interrupt tramite polling.

IL SISTEMA DI EXCEPTION DELL'MC68000

L'MC68000 fornisce una vasta logica di processo in modo Exception, che comprende, oltre ad una serie di interrupt esterni, anche delle Exception generate internamente, in seguito a vari tipi di errori, Trap e così via.

MODI OPERATIVI

Istruzioni che possono modificare il modo operativo

Prima di descrivere i tipi di risposta alle varie Exception, analizziamo i modi operativi dell'MC68000, dal momento che riguardano da vicino questo aspetto. Come abbiamo ricordato in precedenza, **l'MC68000 può operare in modo Supervisore o in modo Utente. Quando è inizializzato mediante un segnale di RESET** inizia a funzionare in modo Supervisore e continua a farlo **finchè non viene eseguita una delle istruzioni seguenti : RTE** (Return From Exception, ritorno da Exception), **MOVE word in SR** (Move to Status Register, trasferimento di un dato al registro di stato), **ANDI word in SR** (AND immediato al registro di stato) ed **EORI word in SR** (OR esclusivo immediato al registro di stato). Nessuna di queste istruzioni causa automaticamente il passaggio al modo Utente; la loro funzione è quella di cambiare lo stato del bit S del registro di stato. **Non appena una di queste istruzioni azzera il bit S, viene selezionato il modo Utente.**

Passaggio da modo Utentee a modo Supervisore

Quando l'MC68000 funziona in modo Utente, l'unica cosa che può provocare un ritorno al modo Supervisore è una Exception. L'elaborazione di un'Exception avviene in modo Supervisore, indipendentemente da quale sia il valore del bit S del registro di stato nel momento in cui essa si verifica. Una volta terminato l'intero processo un'istruzione RTE (Return from Exception) permette di ritornare al modo Utente.

Istruzione privilegiate

Alcune istruzioni, definite "privilegiate", possono essere eseguite soltanto in modo Supervisore. Il tentativo di eseguirle in modo Utente è una cosiddetta "violazione di privilegio" e rappresenta una delle cause di Exception, come vedremo meglio in seguito.

TIPI DI EXCEPTION

L'MC68000 fornisce sempre risposte molto simili ai vari tipi di Exception. Prima di descriverle nei dettagli diamo uno sguardo alle possibili cause di Exception, che, come sappiamo, sono ben più numerose di quelle previste negli altri microprocessori.

Le Exception possono essere suddivise in due categorie, a seconda dell'evento che le ha prodotte:

1. **Exception generate internamente,** dovute all'esecuzione di determinate istruzioni o da errori interni.
2. **Exception generate esternamente:** errori di bus, reset e richieste di interrupt.

Exception Generate Internamente

Errori interni

Le Exception generate internamente le possiamo ulteriormente suddividere in tre categorie: errori interni, istruzioni Trap e la funzione Trace.

Gli errori interni in grado di provocare un'Exception sono:

1. **Errori di indirizzamento.** Ogni tentativo di accedere ad un'istruzione o ad un dato di lunghezza pari ad una word o ad una long word, posto ad un indirizzo dispari, provoca un errore di indirizzamento, dal momento che è necessario utilizzare sempre indirizzi pari.
2. **Violazioni di privilegio.** Come già abbiamo detto, l'esecuzione di alcune istruzioni è possibile soltanto in modo Supervisore e ogni volta che si tenterà di eseguirle in modo Utente viene prodotta un'Exception. Queste istruzioni, definite privilegiate, sono: STOP, RESET, RTE, MOVE in SR, AND (word) Immediato in SR, EOR (word) Immediato in SR, OR (word) Immediato in SR, MOVE USP.

Possibilità di simulare nuove istruzioni

3. **Codici operativi illegali o non implementati.** Il tentativo di eseguire un'istruzione che non appartiene al set di quelle definite per l'MC68000 dà luogo ad una Exception. Inoltre, due particolari sequenze di bit, anzichè illegali, sono definite non implementate: si tratta delle sequenze 1010 e 1111, in corrispondenza dei bit 15-12 di una word d'istruzione. Se il processore preleva codici operativi di questo tipo si verifica un'Exception che consente al programmatore di simulare eventuali nuove istruzioni.

Istruzioni di TRAP

Si possono generare delle Exception anche dall'interno di un programma, mediante le istruzioni di tipo Trap. Infatti, oltre ad un'istruzione TRAP standard, simile all'istruzione System Call dello Z8000, ce ne sono **altre quattro (TRAPV, CHK, DIVS e DIVU) che provocano un'Exception** solo in particolari situazioni, come nel caso di un overflow aritmetico o di una divisione per zero.

Il terzo tipo di Exception generate internamente è dovuto alla funzione Trace. Se il bit T del registro di stato è posto a 1 si verifica un'Exception dopo ogni istruzione. La funzione Trace, permettendo di analizzare, volta per volta, i risultati ottenuti con l'esecuzione di ogni singola istruzione, viene comunemente utilizzata nel debugging dei programmi.

Exception Generate Esternamente

1. **Errori di Bus.** Quando il segnale BERR viene forzato basso da una logica esterna (ed il processore non è bloccato) si verifica un'Exception.
2. **Reset.** Quando il segnale RESET viene attivato da una logica esterna viene generata un'Exception.
3. **Richiesta di interrupt.** È il tipo di Exception più conosciuto ed è prodotto da una logica esterna mediante le tre linee di interrupt (IPL0, IPL1 e IPL2).

Priorità delle Exception

I vari tipi di Exception hanno priorità diverse ed il loro riconoscimento dipende proprio da questo. La tabella seguente elenca i tipi di Exception in base alla loro priorità, indicando anche quando ha inizio la relativa procedura di servizio.

Gruppo	Priorità	Causa di Exception	Tipo di Risposta
0	Alta	Reset Errore di Bus Errore di Indirizzo	Arresta l'attuale ciclo, quindi elabora l'exception
1		Trace Richiesta di Interrupt Codice Operativo Illegale o Non Implementato Violazione di Privilegio	Completa l'attuale istruzione, quindi elabora l'exception
2	Bassa	TRAP, TRAPV CHK Divisione per zero	L'esecuzione dell'istruzione inizia il processo di exception

Situazione che causa l'arresto del processore

Le Exception con priorità più elevata sono quelle causate da un segnale di reset, da un errore di bus oppure da un errore di indirizzamento. Esse interrompono l'esecuzione di un'istruzione, anche durante un ciclo di bus. L'altro gruppo di Exception (Trace, interrupt, istruzioni illegali o non implementate e violazioni di privilegio) consente il completamento dell'istruzione che in quel momento viene eseguita. Le richieste di interrupt prevedono un ulteriore ordine di priorità, che descriveremo più avanti. Le Exception con priorità più bassa sono quelle provocate da istruzioni di tipo Trap, le quali generano un'Exception durante la normale esecuzione di un'istruzione. In questo caso, avremo sempre una stessa priorità, dal momento che è impossibile che vengano eseguite contemporaneamente due istruzioni.

Tabella dei Vettori di Exception

Per l'elaborazione di un'Exception è necessaria una tabella di vettori, costituita da 1024 byte (512 word da 16 bit) e che occupa l'area di

memoria compresa fra 000000_{16} e $0003FF_{16}$. **La Figura 15-1 mostra la tabella dei vettori di Exception,** contenente 256 vettori di quattro byte, ognuno dei quali è un indirizzo a 32 bit, che verrà posto nel contatore di programma nel momento in cui si verifica un'Exception.

Come vedete, alcuni dei vettori servono i tipi di Exception che abbiamo già descritto. Gli altri sono riservati al costruttore e non li dovrete usare in un programma se desiderate mantenere la compatibilità con futuri prodotti hardware o software della Motorola. I primi 64 vettori di Exception hanno degli usi prestabiliti; gli altri 192 sono disponibili per le richieste di interrupt definite dall'utente e dovrebbero essere più che sufficienti per la maggior parte delle applicazioni. (Naturalmente, in questo caso, con "utente" intendiamo il progettista di un microcomputer e non il programmatore in linguaggio assembly). Tuttavia, le locazioni dei primi 64 vettori non sono protette dall'MC68000 e, se necessario, possono essere usate da interrupt esterni.

ELABORAZIONE DELLE EXCEPTION

Procedura di elaborazione

La sequenza generale degli eventi eseguiti dall'MC68000 in risposta ad un'Exception è la stessa, indipendentemente dalla causa che l'ha prodotta. Esistono, tuttavia, delle differenze. Cominciamo con l'esaminare ciò che accade in risposta alle Exception generate internamente.

Risposta ad Exception Generate Internamente

Se la causa di un'Exception è rappresentata dall'attivazione della funzione Trace, da un'istruzione TRAP, da un codice operativo illegale o non implementato oppure da una violazione di privilegio, avremo una sequenza di questo tipo:

1. Il contenuto del registro di stato viene copiato in un registro interno.
2. Il bit S del registro di stato è posto a 1, per operare in modo Supervisore.
3. Il bit T del registro di stato viene azzerato per disattivare il modo Trace.
4. Il valore presente nel contatore di programma viene depositato sullo stack Supervisore.
5. Il contenuto del registro di stato, già salvato in precedenza, viene messo sullo stack Supervisore.
6. Dalla tabella dei vettori di interrupt, viene prelevato il nuovo valore del contatore di programma.

Figura 15-1. Tabella dei Vettori di Exception.

7. Viene eseguita la routine, che inizia alla locazione indicata dal nuovo contenuto del contatore di programma. Si tratterà della routine che voi stessi avete destinato alla gestione di quel particolare tipo di Exception.

Elaborazione di Exception Generate da Errori di Bus e di Indirizzamento.

La risposta dell'MC68000 alle Exception provocate da errori di bus o di indirizzamento prevede tutta una serie di altre operazioni, oltre a quelle descritte nei paragrafi precedenti. Innanzitutto, entrambi questi errori causano la fine immediata di un ciclo di bus. Le fasi successive sono le seguenti:

1. Il contenuto del registro di stato viene copiato in un registro interno.
2. Il bit S del registro di stato è posto a 1, per selezionare il modo Supervisore.
3. Il bit T del registro di stato viene azzerato per disattivare il modo Trace.
4. Il contenuto del contatore di programma viene posto sullo stack Supervisore.
5. Il registro di stato, già copiato in precedenza, viene messo sullo stack Supervisore.
6. Il registro istruzioni contenente la prima word dell'istruzione eseguita quando si è verificato l'errore di bus, viene memorizzato sullo stack Supervisore.
7. L'indirizzo a 32 bit utilizzato durante il ciclo di bus interrotto, viene anch'esso posto sullo stack Supervisore.
8. Sullo stack Supervisore viene deposta anche una word che indica il tipo di ciclo che era in corso al momento dell'errore.
9. Dalla posizione della tabella dei vettori, relativa agli errori di bus (o a quelli di indirizzamento), l'MC68000 preleva il nuovo valore del contatore di programma.
10. L'esecuzione delle istruzioni riprende dalla locazione indicata dal contatore di programma.

La Figura 15-2 mostra l'ordine in cui tutti questi dati sono messi sullo stack Supervisore, in risposta ad un'Exception causata da un errore di bus o di indirizzamento. Il vecchio valore del contatore di programma viene incrementato relativamente all'indirizzo della prima word dell'istruzione eseguita nel momento in cui si è verificato l'errore. L'entità dell'incremento varia da 2 a 10 byte, a seconda della lunghezza dell'istruzione e dell'indirizzo utilizzato come operando.

Se l'errore si verifica durante la fase in cui viene prelevata l'istruzione successiva, l'MC68000 salva il contatore di programma con l'indirizzo dell'istruzione eseguita in quel momento, anche se si tratta di un'istruzione di salto, di diramazione o di ritorno. Questa

caratteristica, assente nella maggioranza dei computer, facilita l'individuazione di molti errori.

Come si può osservare nella Figura 15-2, **i cinque bit meno significativi dell'ultima word messa sullo stack forniscono le informazioni relative al tipo di accesso che era in corso quando è avvenuto l'errore di bus o di indirizzamento.** I tre bit meno significativi sono una copia dei codici di funzione durante il ciclo di bus interrotto. Il bit 3 indica il tipo di elaborazione in corso al momento dell'errore: vale 1 nel caso di un'Exception del Gruppo 0 o del Gruppo 1 ed è azzerato per Exception del Gruppo 2 e nel caso si tratti di una normale istruzione (cfr. la precedente tabella con l'indicazione delle priorità). Il bit 4 indica se, al momento dell'errore, era in corso un ciclo di lettura (1) o di scrittura (0). Se si verifica un errore durante l'elaborazione di un'Exception causata da un precedente errore di bus o di indirizzamnento oppure da un'operazione di reset, l'MC68000 si pone indefinitamente nello stato Halt.

Codice Funzione	Tipo di ciclo
0	Non Assegnato
1	Dati Utente
2	Programma Utente
3	Non Assegnato
4	Non Assegnato
5	Dati Supervisore
6	Programma Supervisore
7	Riconoscimento Interrupt

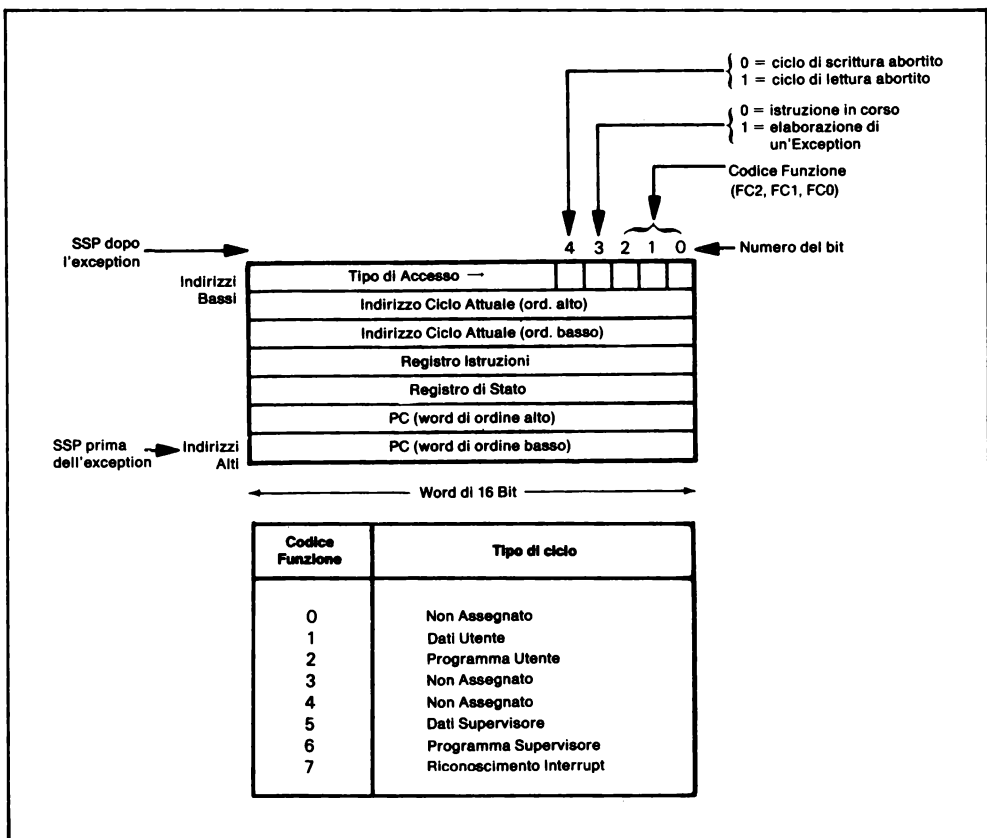



Figura 15-2. Stack di Sistema dopo un Exception dovuta ad un Errore di Bus o di Indirizzamento

Tutte le informazioni messe sullo stack Supervisore durante una Exception dovuta ad un errore di bus o di indirizzamento hanno lo scopo di fornire un aiuto nell'analisi delle possibili cause di errore. Entrambi questi errori comportano una grave alterazione del sistema ed è praticamente impossibile riprendere la normale esecuzione del programma.

Elaborazione di Exception causate da Reset

Il trattamento di un'Exception determinata da un segnale di RESET presenta caratteristiche particolari. Ecco le varie fasi:

1. Il bit S del registro di stato è posto a 1, per selezionare il modo Supervisore.
2. Il bit T del registro di stato viene azzerato per disattivare il modo Trace.
3. I tre bit della maschera di interrupt del registro di stato sono posti a 1, per indicare una priorità pari a 7.
4. Nel puntatore allo stack Supervisore viene trasferito il contenuto dei primi quattro byte di memoria (indirizzi 000000-000003).
5. Nel contatore di programma vengono posti i quattro byte di memoria successivi (000004-000007).
6. L'esecuzione delle istruzioni avviene a partire all'indirizzo presente nel contatore di programma, che è, poi, l'indirizzo iniziale della routine che voi avete previsto debba essere eseguita al momento dell'accensione del sistema o dopo un segnale di RESET.

Elaborazione di Exception causate da una Richiesta di Interrupt

L'ultimo tipo di risposta ad un'Exception che prendiamo in esame è quello determinato da una richiesta di interrupt. Un dispositivo esterno provoca un interrupt inviando, sui corrispondenti ingressi, un codice con il relativo livello di priorità. L'MC68000 lo confronta con il valore indicato dai tre bit della maschera di interrupt del registro di stato. Se il livello di priorità di un interrupt è inferiore od uguale a quello specificato dalla maschera, la richiesta non è accettata dall'MC68000. **Se la priorità è superiore a quella indicata dalla maschera (oppure il suo valore è sette), allora l'interrupt viene riconosciuto.** L'MC68000 risponde alla richiesta di interrupt non appena ha completato l'istruzione che sta eseguendo. Una volta terminata l'istruzione, questo è ciò che accade:

1. Il contenuto del registro di stato viene salvato internamente.
2. Il bit S del registro di stato viene messo a 1, per operare in modo Supervisore.

3. il bit T del registro di stato è azzerato per disattivare la funzione Trace.
4. La maschera di interrupt assume, mediante la modifica dei relativi bit del registro di stato, un valore uguale alla priorità dell'interrupt ricevuto. Questo consente di gestire un interrupt senza la possibilità di ulteriori interruzioni dovute ad eventi dello stesso livello di priorità o di priorità inferiore.
5. L'MC68000 esegue, quindi, un ciclo di bus destinato al riconoscimento dell'interrupt. Questo ciclo riveste una duplice funzione: innanzitutto, il processore informa il dispositivo richiedente che sta servendo l'interrupt inviatogli e, in secondo luogo, va a prelevare il byte del cosiddetto vettore di Exception dallo stesso dispositivo richiedente.
6. Il contenuto del contatore di programma viene messo sullo stack Supervisore.
7. Il contenuto del registro di stato, salvato in precedenza, viene messo sullo stack Supervisore.
8. Sul contatore di programma sono caricati quattro byte letti dalla tabella dei vettori a partire dalla locazione indicata dal byte del vettore di Exception.

Una volta effettuata questa operazione, l'esecuzione continua a partire dalla locazione indicata dal nuovo valore presente nel contatore di programma: essa conterrà la prima istruzione della routine che avrete destinato alla gestione della richiesta di interrupt, proveniente da una determinata periferica.

Risposta all'Interrupt in modo Autovettore

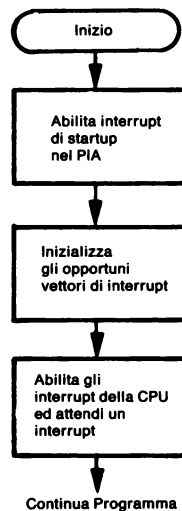
Una variante nel modo in cui viene gestita una richiesta di interrupt è la risposta in modo autovettore. Come potete osservare dalla figura 15-1, nella tabella dei vettori di Exception sette locazioni sono riservate agli autovettori corrispondenti ai sette livelli di priorità degli interrupt. Questi vettori sono utilizzati quando il dispositivo richiedente risponde al ciclo di bus che indica l'avvenuto riconoscimento di un interrupt attivando l'ingresso VPA (Valid Peripheral Address), invece di fornire un byte con la posizione del vettore. Il processore si servirà, allora, dell'autovettore corrispondente al livello di priorità dell'interrupt per ottenere un nuovo valore da porre nel contatore di programma. Questo tipo di risposta, in modo autovettore, è stata fornita soprattutto per emulare la sequenza di interrupt prevista dai dispositivi periferici della famiglia 6800. Naturalmente, questa possibiità potrà essere sfruttata anche con dispositivi diversi, nel caso che ciò possa rivelarsi utile.

ESEMPI DI PROGRAMMAZIONE

15-1. Startup (avviamento)

Scopo: Accendere il computer ed attendere un interrupt proveniente dal PIA prima di iniziare l'attività vera e propria.

Quando un sistema MC68000 è acceso, il processore viene resettato e comincia il processo di inizializzazione. Con RESET il processore è posto in stato supervisore e la maschera delle priorità di interrupt è messa a 1 per inibire tutti gli interrupt eccetto quelli di livello 7. Il puntatore allo stack di supervisore è caricato con le prime 2 word del vettore di Exception reset alla locazione di memoria 0. Il contatore di programma è caricato con le due word successive e l'esecuzione inizia con l'istruzione il cui indirizzo è contenuto nel contatore di programma.

Diagramma di Flusso 15-1

Se nella ROM (memoria di sola lettura) o nella ROM programmabile (PROM) si trovasse questo programma, nel momento in cui si verifica il RESET dovuto all'accensione del sistema, sullo stack Supervisore verrebbe posto il valore 5100 e nel contatore di programma il valore 4000, l'indirizzo della routine di avviamento. Dopo che il bit Supervisore e quelli della maschera di interrupt, nel registro di stato, sono stati messi a 1, sarà eseguito il programma che inizia alla locazione 4000.

Programma 15-1

```
00004000:                   POWER    EQU      $4000
00004600:                   SERVICE  EQU      $4600
00005100:                   STACK    EQU      $5100
00006000:                   DATA     EQU      $6000
                            ;
0003FF40:                   PIADDA   EQU      $3FF40         REGISTRO DIREZIONE DATI A
0003FF40:                   PIADA    EQU      $3FF40         REGISTRO DATI A
0003FF44:                   PIACA    EQU      $3FF44         REGISTRO DI CONTROLLO A
00000005:                   PIA_EN   EQU      $05            ABILITAZIONE INTERRUPT
00002000:                   IMSK0    EQU      $2000          SUPERVISORE/INTERRUPT LIV. 0
00000064:                   SVECTOR  EQU      $64            IND. VETTORE D'INTERRUPT
                            ;
                                     ORG      0

00000000:                            DC.L     STACK          INDIRIZZO DELLO STACK
00000004:                            DC.L     PGM15_1        INDIRIZZO DEL PROGRAMMA DI RESET
                            ;
                                     ORG      POWER

00004000: 13FC 0005
00004004: 0003 FF44         PGM15_1  MOVE.B   #PIA_EN,PIACA  ABILITA INTERRUPT PIA STARTUP
00004008: 21FC 0000
0000400C: 4600 0064                  MOVE.L   #STARTUP,SVECTOR INIZIALIZZA VETTORE PIA
00004010: 4E72 2000                  STOP     #IMSK0         ABILITA INTERRUPT E ATTENDI
                            ;
                            ;   *    ROUTINE DI SERVIZIO INTERRUPT DI STARTUP
                            ;
                                     ORG      SERVICE

00004600: 4A39 0003
00004604: FF40              STARTUP  TST.B    PIADA          AZZERA INTERRUPT DI STARTUP
00004606: 4E73                       RTE                     RITORNA ALLA ROUTINE INTERROTTA
                            ;

                                     END      PGM15_1
```

A differenza degli altri vettori di Exception, quello di reset deve trovarsi nella ROM o nella PROM. La stessa cosa vale anche per la routine che dovrà essere eseguita. Bisognerà accertarsi che i valori presenti nel vettore di reset e destinati al puntatore di stack ed al contatore di programma, siano indirizzi validi della ROM e della RAM.

Tutti gli altri vettori di Exception possono trovarsi nella RAM o nella ROM: è in fase di progettazione che bisogna scegliere la posizione che più si adatta alle caratteristiche del sistema. Dato che, nel nostro esempio, i vettori di Exception si trovano nella RAM, devono essere inizializzati con gli indirizzi delle corrispondenti routine di servizio, prima che si verifichi una qualunque Exception.

L'istruzione MOVE.L #STARTUP,SVECTOR inizializza il vettore di Exception associato con il PIA. Nel nostro esempio, gli interrupt provenienti dal PIA hanno una priorità bassa ed è stato loro assegnato il livello 1. Dal momento che il PIA (ed anche l'ACIA) non forniscono il numero di un vettore, i loro interrupt sono gestiti dall'MC68000 in modo autovettore. Come è indicato nella Figura 15-1, gli autovettori iniziano all'indirizzo 64, che corrisponde, appunto, all'interrupt autovettore di livello 1. Questo è il vettore che conterrà l'indirizzo iniziale della nostra routine di servizio.

Se dimentichiamo di inizializzare un vettore di Exception, il processore se ne servirà ugualmente per stabilire l'indirizzo iniziale della routine per la gestione dell'Exception, anche se si tratta di un indirizzo non valido. **Sarà necessario, perciò, inizializzare i vettori di Exception, proprio come si inizializzano certi dati di un programma prima di utilizzarli.**

Oltre a definire il vettore di Exception, l'unica altra funzione del programma è quella di abilitare l'interrupt proveniente dal PIA al

momento dell'accensione del sistema, ponendo a 1 il bit 0 del registro di controllo e,quindi, abilitando l'intero sistema di interrupt dell'MC68000.

In questo modo, il programma è pronto per l'interrupt di startup. Invece di attenderlo, eseguendo un ciclo senza fine del tipo LOOP: JMP LOOP, ci possiamo servire dell'istruzione STOP, che costringe il processore a sospendere l'esecuzione delle istruzioni e ad attendere un interrupt o un'altra Exception (TRACE o RESET). L'istruzione STOP consente anche di cambiare il valore della maschera di interrupt, in quanto la word successiva allo STOP viene caricata nel registro di stato. Affinchè gli interrupt siano riconosciuti, è necessario portare il livello di priorità dal valore 7 (prodotto durante il RESET) al valore 0 (immediatamente inferiore al livello 1, assegnato al PIA). Una priorità 0 consente al processore di riconoscere interrupt di qualsiasi livello. Inoltre, la word utilizzata come operando dell'istruzione STOP dovrà avere il bit corrispondente al bit S (Supervisore) del registro di stato posto a 1.

L'istruzione STOP

L'istruzione STOP è una delle poche istruzioni dell'MC68000 che possono essere eseguite solo nel modo Supervisore. Se eseguita nel modo Utente provoca un'Exception dovuta a violazione di privilegio. (Generalmente, sono istruzioni privilegiate quelle che cambiano il livello di interrupt del processore, lo fanno passare dal modo Supervisore a quello Utente, o viceversa, oppure alterano il puntatore allo stack Utente).

Con l'arrivo di un interrupt dal PIA inizia il processo di Exception. Per prima cosa, il contenuto dell'intero registro di stato viene salvato alla sommità dello stack Supervisore, seguito dal valore presente nel contatore di programma, che è, poi, l'indirizzo della prossima istruzione (in questo caso, quella immediatamente successiva all'istruzione STOP). Viene selezionato il modo Supervisore e la maschera di interrupt assume lo stesso valore della priorità dell'interrupt appena ricevuto. Quindi il programma va a prendersi l'indirizzo della routine di servizio dal corrispondente vettore. Dal momento che si tratta di un interrupt autovettore di livello 1, il vettore si troverà all'indirizzo 64.

Al momento dell'ingresso nella routine di servizio dell'interrupt, posta alla locazione STARTUP, il livello di priorità è 1 ed il processore si trova in modo Supervisore. Il valore della maschera di interrupt è stato cambiato, affinchè altri interrupt con una priorità di livello 1 non interrompano l'attività del processore. Cosa sarebbe accaduto se l'istruzione STOP avesse posto il livello di priorità a 1?

Disattivazione di un interrupt

La routine di servizio disattiva l'interrupt di startup leggendo l'opportuno registro dati del PIA. Questa operazione è necessaria, anche se non vi è un trasferimento di dati, poichè, altrimenti, l'interrupt rimarrebbe attivo, causando una nuova interruzione, non appena sono riabilitati gli interrupt di livello 1.

Per disattivare l'interrupt viene impiegata l'istruzione TST,che non modifica nessun registro ad eccezione di quello del codice di condizione. **In questo caso, non ci interessa salvare nessun registro dati o indirizzi, ma se ciò fosse necessario, bisogna provvedere al**

L'istruzione RTE

salvataggio in fase di ingresso nella routine, ripristinandoli al momento dell'uscita.

L'istruzione RTE restituisce il controllo al programma che è stato interrotto e, più esattamente, all'istruzione successiva a quella di STOP. Durante questa fase al bit S e alla maschera di priorità sono restituiti i valori che avevano prima dell'interrupt, prelevando il registro di stato, precedentemente salvato sullo stack. Quindi viene ripristinato il precedente valore del contatore di programma, prelevato anch'esso dallo stack. RTE non modifica nessun altro registro, ad eccezione del contatore di programma e del registro di stato. Al pari dello STOP, RTE è un'istruzione privilegiata e può essere eseguita solo nel modo Supervisore.

Questo programma presuppone che non siano possibili altri interrupt di livello 1. Qualora, invece, questi potessero verificarsi sarà necessario aggiungere una routine di polling a quella già riservata alla gestione dell'interrupt, modificando, di conseguenza, anche il programma principale. Siete in grado di farlo?

15-2. Un interrupt da tastiera

Scopo: Il programma principale azzera la variabile FLAG, alla locazione di memoria 6000, ed attende un interrupt dalla tastiera. La routine di sevizio dell'interrupt pone FLAG a 1 e mette il dato letto dalla tastiera nella variabile KEY, alla locazione 6001.

Problema Campione:

	Dato Tastiera = 43	
Risultato:	FLAG - (6000) = 01	Flag indicante la disponibilità di un nuovo dato
	KEY - (6001) = 43	Dato Tastiera

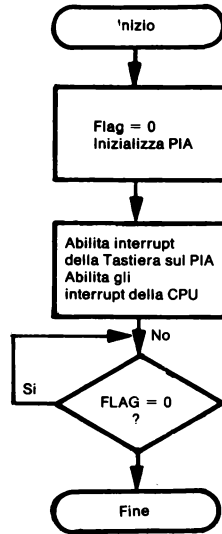
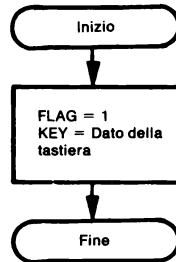
Prima di abilitare gli interrupt bisogna inizializzare il PIA, definendo le direzioni delle porte e delle linee di controllo ed indicando quale tipo di transizioni, sugli ingressi di strobe, dovranno essere riconosciute.

Il programma principale azzera il flag che indica la disponibilità di un dato (FLAG) ed attende semplicemente che la routine di servizio dell'interrupt lo metta a 1. Il programma principale e la routine di servizio comunicano attraverso due indirizzi di memoria fissi:

la variabile FLAG indica se è stato ricevuto un nuovo dato dalla tastiera;

la variabile KEY è un buffer dati di un'unica locazione, che dovrà contenere il valore ricevuto dalla tastiera.

Diagramma di Flusso 15-2

Programma principale:

Routine di Servizio degli interrupt:

Programma 15-2a

```
00004000:                PROGRAM  EQU    $4000
00004600:                INT_25   EQU    $4600
00006000:                DATA     EQU    $6000
                         ;
0003FF40:                PIADDA   EQU    $3FF40      REGISTRO DIREZIONE DATI A
0003FF40:                PIADA    EQU    $3FF40      REGISTRO DATI A
0003FF44:                PIACA    EQU    $3FF44      REGISTRO DI CONTROLLO A
00000005:                PIA_EN   EQU    $05         ABILITAZIONE INTERRUPT
00002000:                IMSK0    EQU    $2000       SUPERVISORE/INTERRUPT LIV. 0
                         ;
                                  ORG    DATA
00006000:                FLAG     DS.B   1           FLAG DATO DISPONIBILE
00006001:                KEY      DS.B   1           TASTO PREMUTO
                         ;
                                  ORG    PROGRAM
00004000: 4238 6000      PGM15_2A CLR.B  FLAG        AZZERA FLAG DATO DISPONIBILE
00004004: 4239 0003
00004008: FF44                    CLR.B  PIACA       IND. REG. DIREZIONE DATI
0000400A: 4239 0003
0000400E: FF40                    CLR.B  PIADDA      TUTTE LE LINEE SONO INGRESSI
                         ;
```

```
00004010: 13FC 0005
00004014: 0003 FF44          MOVE.B  #PIA_EN,PIACA   ABILITA INTERRUPT PIA TASTIERA
00004018: 46FC 2000          MOVE    #IMSK0,SR       ABILITA TUTTI GLI INTERRUPT
0000401C: 4A38 6000   WTRDY  TST.B   FLAG            C'E' UN DATO DALLA TASTIERA?
00004020: 67FA               BEQ     WTRDY           NO, ATTENDI
                      ;
00004022: 4E75               RTS
                      ;
                      ;   *  ROUTINE DI SERVIZIO DEGLI INTERRUPT
                      ;
                             ORG     INT_25

00004600: 11FC 0001
00004604: 6000               MOVE.B  #1,FLAG         PONI A 1 FLAG DATO DISPONIBILE
00004606: 11F9 0003
0000460A: FF40 6001          MOVE.B  PIADA,KEY       SALVA IL DATO DELLA TASTIERA
0000460E: 4E73               RTE                     RITORNA ALLA ROUTINE INTERROTTA

                             END     PGM15_2A
```

Descrizione del programma 15-2a

Si osservi la somiglianza fra la locazione FLAG ed il bit di stato del registro di controllo del PIA. Non è necessario che il programma controlli il bit 7 del registro di controllo del PIA, perchè esiste un collegamento hardware diretto (interrupt) fra questo bit e la CPU. Naturalmente, si presuppone anche che la tastiera sia l'unica causa possibile di interrupt.

Diversamente dall'esempio precedente, non ci serviamo questa volta dell'istruzione privilegiata STOP, ma controlliamo la variabile FLAG per stabilire se si è verificato o meno un interrupt. Non dimenticate, comunque, che l'istruzione STOP, oltre ad attendere che si verifichi un interrupt, provvede anche a mettere nel registro di stato il livello di priorità desiderato. Nel Programma 15-2a per selezionare il livello di interrupt utilizziamo l'istruzione MOVE verso il registro di stato (MOVE #IMSK0,SR). La word (in questo caso $2000) successiva a quella del codice operativo definisce il nuovo livello di priorità, oltre ai vari codici di condizione del registro di stato. Anche questa è un'istruzione privilegiata.

Si presenta, in certi casi, la necessità che il processore riconosca interrupt di livello inferiore a quello indicato dalla maschera di priorità del registro di stato. Prima di abilitare interrupt di livello più basso, dovremo salvare l'attuale valore della maschera, per poi ripristinarlo una volta servito l'interrupt. Per salvare la maschera di interrupt (insieme con tutto il registro di stato) possiamo utilizzare l'istruzione MOVE dal registro di stato, che non è un'istruzione privilegiata.

Non bisogna dimenticare che, al momento dell'ingresso nella routine di servizio, la CPU ha già provveduto a modificare la maschera di interrupt, assegnandole lo stesso livello di priorità dell'interrupt che in quel momento è impegnata a gestire. Questo serve ad inibire ulteriori interrupt dello stesso livello o di livello più basso: verranno accettati solo interrupt di livello superiore.

L'istruzione RTE, alla fine della routine di servizio, restituisce il controllo al programma principale. Volendo trasferire il controllo ad un'altra parte della memoria (magari ad una routine per la gestione di eventuali errori) bisogna cambiare il valore del contatore di programma presente sullo stack Supervisore, ricorrendo ad uno dei metodi descritti in precedenza. RTE ripristina anche il valore della maschera relativa alla priorità degli interrupt.

In questo esempio, non ci serviamo dei registri per trasferire i parametri ed i risultati. Cambiando i valori dei registri possiamo

sempre interferire con l'esecuzione del programma principale. Nella maggior parte delle applicazioni il programma principale fa un notevole uso dei registri ed eventuali cambiamenti casuali finirebbero per provocare un disastro. Nella migliore delle ipotesi cambiando i registri si creano degli inutili vincoli, che rendono impossibile una loro eventuale utilizzazione successiva per scopi diversi.

Non è necessario che la routine di servizio riabiliti esplicitamente gli interrupt in quanto RTE ripristina automaticamente il vecchio registro di stato assieme al precedente livello di priorità. In sostanza, se non vogliamo ripristinare la precedente priorità, sarà necessario intervenire sul valore presente sullo stack.

Ci possiamo servire dello stack anche per salvare e, ripristinare successivamente, altri dati (come i contenuti di una locazione di memoria). Questo metodo si può espandere all'infinito (fin quando c'è della RAM disponibile per lo stack), poichè routine di servizio nidificate non distruggono i dati salvati dalle routine precedenti.

Riempire un Buffer mediante Interrupt

Una soluzione alternativa è quella di far sì che la routine di servizio dell'interrupt ponga a 1 FLAG solo dopo aver ricevuto un'intera linea di testo (una stringa di caratteri che termina con un ritono di carrello). In questo caso, usiamo FLAG per indicare la fine della riga e le locazioni di memoria 6002 e 6003 conterranno il puntatore ad un buffer (POINTER). Si presuppone che il buffer abbia inizio alla locazione 6004.

Programma 15-2b

```
00004000:                  PROGRAM  EQU     $4000
00004600:                  INT_25   EQU     $4600
00006000:                  DATA     EQU     $6000
                           ;
0003FF40:                  PIADDA   EQU     $3FF40          REGISTRO DIREZIONE DATI A
0003FF40:                  PIADA    EQU     $3FF40          REGISTRO DATI A
0003FF44:                  PIACA    EQU     $3FF44          REGISTRO DI CONTROLLO A
00000005:                  PIA_EN   EQU     $05             ABILITAZIONE INTERRUPT
00002000:                  IMSK0    EQU     $2000           SUPERVISORE/INTERRUPT LIV. 0
0000000D:                  CR       EQU     $0D             RITORNO CARRELLO
                           ;
                                    ORG     DATA

00006000:                  FLAG     DS.B    1               FLAG DI FINE RIGA
00006001:                           DS.B    1
00006002:                  POINTER  DS.W    1               PUNTATORE FINE BUFFER + 1
00006004:                  BUFFER   DS.B    80              BUFFER DI INPUT
                           ;
                                    ORG     PROGRAM

00004000: 4238 6000        PGM15_2B CLR.B   FLAG            AZZERA FLAG DATO DISPONIBILE
00004004: 31FC 6004
00004008: 6002                      MOVE.W  #BUFFER,POINTER INIZIALIZZA PUNTATORE
0000400A: 4239 0003
0000400E: FF44                      CLR.B   PIACA           IND. REG. DIREZIONE DATI
00004010: 4239 0003
00004014: FF40                      CLR.B   PIADDA          TUTTE LE LINEE SONO INGRESSI
00004016: 13FC 0005
0000401A: 0003 FF44                 MOVE.B  #PIA_EN,PIACA   ABILITA INTERRUPT PIA TASTIERA
0000401E: 46FC 2000                 MOVE    #IMSK0,SR       ABILITA TUTTI GLI INTERRUPT
00004022: 4A38 6000        WTRDY    TST.B   FLAG            C'E' UN DATO DALLA TASTIERA?
00004026: 67FA                      BEQ     WTRDY           NO, ATTENDI
                           ;
00004028: 4E75                      RTS
                           ;
                           ; *      ROUTINE DI SERVIZIO DEGLI INTERRUPT
                           ;
                                    ORG     INT_25

00004600: 2F08                      MOVE.L  A0,-(SP)        METTI A0 SULLO STACK SUPERVISORE
00004602: 3078 6002                 MOVE.W  POINTER,A0      PRENDI NUOVO PUNT. BUFFER
00004606: 10F9 0003
0000460A: FF40                      MOVE.B  PIADA,(A0)+     SALVA IL DATO DEL TASTO NEL BUFFER
```

```
0000460C: 0C28 000D
00004610: FFFF                  CMPI.B  #CR,-1(A0)      E' UN RITONO CARRELLO?
00004612: 6606                  BNE.S   DONE            NO, RITORNA
00004614: 11FC 0001
00004618: 6000                  MOVE.B  #1,FLAG         PONI A 1 FLAG DI FINE RIGA
0000461A: 31C8 6002     DONE    MOVE.W  A0,POINTER      AGGIORNA PUNTATORE BUFFER
0000461E: 205F                  MOVE.L  (SP)+,A0        RIPRISTINA REGISTRO A0
00004620: 4E73                  RTE                     RITORNA ALLA ROUTINE INTERROTTA

                                END     PGM15_2B
```

Descrizione del programma 15-2b

Questo programma riempie un buffer, che inizia alla locazione di memoria 6004, finchè non riceve un carattere di ritorno carrello (CR). POINTER contiene il puntatore al buffer, che la routine di servizio dell'interrupt provvede ad incrementare (autoincremento) ogni volta che lo utilizza.

Nelle applicazioni reali, la CPU potrebbe svolgere altre funzioni fra un interrupt e l'altro. Ad esempio, redarre, spostare o trasmettere una linea da un buffer, mentre la routine di servizio dell'interrupt ne riempie un altro. È il cosiddetto metodo della doppia bufferizzazione. Il programma principale deve solo accertarsi che non si esaurisca il buffer utilizzato dalla routine d'interrupt.

Metodo della doppia bufferizzazione

Un altro metodo potrebbe essere quello di usare FLAG come contatore. Il valore in essa contenuto indicherà, allora, al programma principale quanti byte sono stati ricevuti, senza che questo debba contarli. Possiamo stabilire, anche, di occuparci del buffer, solamente quando esso contiene un certo numero di caratteri. Durante ogni operazione di input la routine di servizio si limiterà all'incremento del contatore e del puntatore del buffer.

Le routine per il servizio degli interrupt vengono richiamate in modo del tutto casuale, per la natura stessa degli interrupt, e non possiamo mai sapere quali registri stava usando il programma quando ha subito l'interruzione. **Per prevenire modifiche accidentali, è necessario salvare e ripristinare il contenuto di tutti i registri utilizzati dalla routine di servizio.** L'istruzione MOVEM, di cui abbiamo parlato in precedenza, si rivela particolarmente adatta a questo scopo.

15-3. Un interrupt da stampante

Scopo: Il programma principale azzera la variabile FLAG, alla locazione di memoria 6000, ed attende un interrupt dalla stampante, che, in questo modo, conferma la sua disponibilità a ricevere una dato. La routine di servizio dell'interrupt pone a 1 FLAG ed invia alla stampante il contenuto della variabile CHAR, alla locazione 6001.

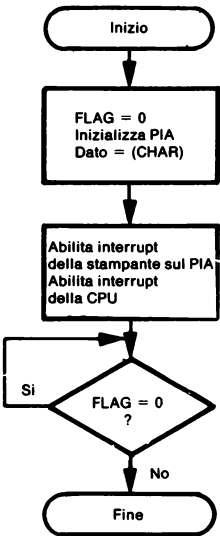
Problema Campione:

CHAR - (6000) = 51

Risultato: FLAG - (6000) = 01 indica che l'ultimo dato è stato inviato

La stampante riceve un 5116(ASCII Q), quando è pronta.

Diagramma di Flusso 15-3a

Programma principale:

Routine di Servizio degli Interrupt:

Programma 15-3a

```
00004000:          PROGRAM  EQU    $4000
00004600:          INT_25   EQU    $4600
00006000:          DATA     EQU    $6000
                   ;
0003FF40:          PIADDA   EQU    $3FF40      REGISTRO DIREZIONE DATI A
000000FF:          DLOUT    EQU    $FF         TUTTE LE LINEE SONO USCITE
0003FF40:          PIADA    EQU    $3FF40      REGISTRO DATI A
0003FF44:          PIACA    EQU    $3FF44      REGISTRO DI CONTROLLO A
00000005:          PIA_EN   EQU    $05         ABILITAZIONE INTERRUPT
00002000:          IMSK0    EQU    $2000       SUPERVISORE/INTERRUPT LIV. 0
                   ;
                            ORG    DATA

00006000:          FLAG     DS.B   1           FLAG DATO ACCETTATO
00006001:          CHAR     DS.B   1           OUTPUT ALLA STAMPANTE
                   ;
                            ORG    PROGRAM
```

```
00004000: 4238 6000     PGM15_3A CLR.B   FLAG            AZZERA FLAG DATO ACCETTATO
00004004: 4239 0003              CLR.B   PIACA           IND. REG. DIREZIONE DATI
00004008: FF44
                        ;
0000400A: 13FC 00FF              MOVE.B  #DLOUT,PIADDA   TUTTE LE LINEE SONO USCITE
0000400E: 0003 FF40
00004012: 13FC 0005              MOVE.B  #PIA_EN,PIACA   ABILITA INTERRUP PIA STAMPANTE
00004016: 0003 FF44
0000401A: 46FC 2000              MOVE    #IMSK0,SR       ABILITA TUTTI GLI INTERRUPT
0000401E: 4A38 6000     WTACK    TST.B   FLAG            INVIATO UN DATO ALLA STAMPANTE?
00004022: 67FA                   BEQ     WTACK           NO, ATTENDI
                        ;
00004024: 4E75                   RTS
                        ;
                        ;  *     ROUTINE DI SERVIZIO DEGLI INTERRUPT
                        ;
                                 ORG     INT_25

00004600: 11FC 0001              MOVE.B  #1,FLAG         PONI A 1 FLAG DATO ACCETTATO
00004604: 6000
00004606: 4A39 0003              TST.B   PIADA           AZZERA INTERRUPT STAMPANTE
0000460A: FF40
0000460C: 13F8 6001              MOVE.B  CHAR,PIADA      INVIA IL DATO ALLA STAMPANTE
00004610: 0003 FF40
00004614: 4E73                   RTE                     RITORNA ALLA ROUTINE INTERROTTA

                                 END     PGM15_3A
```

Le sole differenze, rispetto alle routine di interrupt da tastiera, sono rappresentate dal significato del flag, dalla direzione di trasferimento dei dati e dal fatto che l'istruzione TST.B deve azzerare il bit 7 del registro di controllo del PIA. Tutte le operazioni di input, ma non quelle di output, azzerano automaticamente questo bit.

Quando FLAG vale zero, significa che la CPU ha un dato disponibile da inviare alla stampante. Quando la routine di servizio dell'interrupt mette a 1 il flag, il programma principale sa che il dato è stato inviato. Perciò, FLAG serve a indicare quando la stampante è pronta a ricevere un dato e quando un dato è stato ricevuto.

Sarebbe opportuno inserire un'operazione di lettura, all'inizio del programma principale, per eliminare la possibilità di interrupt casuali. A tale scopo, si possono impiegare le istruzioni MOVE.B PIADA,D0 oppure TST.B PIADA, ricordando, però, di metterle dopo l'istruzione che seleziona il registro direzione dati, ma prima di quella che abilita gli interrupt della CPU.

Svuotare un Buffer mediante Interrupt

Come nell'esempio relativo alla tastiera, anche in questo caso **la routine di servizio dell'interrupt potrebbe mettere a 1 il flag che indica l'invio del dato solo dopo aver inviato alla stampante un'intera linea di dati,** con un ritorno di carrello alla fine. Useremo ancora FLAG per indicare la fine della linea e le locazioni di memoria 6002 e 6003 per contenere il puntatore del buffer, che si presuppone abbia inizio alla locazione 6004.

Programma 15-3b

```
00004000:               PROGRAM  EQU     $4000
00004600:               INT_25   EQU     $4600
00006000:               DATA     EQU     $6000
                        ;
0003FF40:               PIADDA   EQU     $3FF40          REGISTRO DIREZIONE DATI A
000000FF:               DLOUT    EQU     $FF             TUTTE LE LINEE SONO USCITE
0003FF40:               PIADA    EQU     $3FF40          REGISTRO DATI A
0003FF44:               PIACA    EQU     $3FF44          REGISTRO DI CONTROLLO A
00000005:               PIA_EN   EQU     $05             ABILITAZIONE INTERRUPT
00002000:               IMSK0    EQU     $2000           SUPERVISORE/INTERRUPT LIV. 0
0000000D:               CR       EQU     $0D             RITORNO CARRELLO
                        ;
                                 ORG     DATA
00006000:               FLAG     DS.B    1               FLAG DI FINE RIGA
```

```
00006001:                      DS.B    1
00006002:             POINTER  DS.W    1                PUNTATORE FINE BUFFER + 1
00006004:             BUFFER   DS.B    80               BUFFER DI INPUT
                      ;
                               ORG     PROGRAM

00004000: 4238 6000   PGM15_3B CLR.B   FLAG             AZZERA FLAG FINE RIGA
00004004: 31FC 6004
00004008: 6002                 MOVE.W  #BUFFER,POINTER  INIZIALIZZA PUNTATORE
0000400A: 4239 0003
0000400E: FF44                 CLR.B   PIACA            IND. REG. DIREZIONE DATI
                      ;
00004010: 13FC 00FF
00004014: 0003 FF40            MOVE.B  #DLOUT,PIADDA    TUTTE LE LINEE SONO USCITE
00004018: 13FC 0005
0000401C: 0003 FF44            MOVE.B  #PIA_EN,PIACA    ABILITA INTERRUP PIA STAMPANTE
00004020: 46FC 2000            MOVE    #IMSK0,SR        ABILITA TUTTI GLI INTERRUPT
00004024: 4A38 6000   WTEOL    TST.B   FLAG             STAMPATA TUTTA LA RIGA?
00004028: 67FA                 BEQ     WTEOL            NO, ATTENDI
                      ;
0000402A: 4E75                 RTS
                      ;
                      ;  *     ROUTINE DI SERVIZIO DEGLI INTERRUPT
                      ;
                               ORG     INT_25

00004600: 2F08                 MOVE.L  A0,-(SP)         METTI A0 SULLO STACK SUPERVISORE
00004602: 4A39 0003
00004606: FF40                 TST.B   PIADA            AZZERA INTERRUPT STAMPANTE
00004608: 3078 6002            MOVE.W  POINTER,A0       PRENDI PUNT.NUOVO INGRESSO
0000460C: 13D8 0003
00004610: FF40                 MOVE.B  (A0)+,PIADA      INVIA NUOVO CAR. A STAMPANTE
00004612: 0C28 000D
00004616: FFFF                 CMPI.B  #CR,-1(A0)       L'ULTIMO CAR.ERA UN RITORNO CARRELLO?
00004618: 6606                 BNE.S   DONE             NO,RITORNA
0000461A: 11FC 0001
0000461E: 6000                 MOVE.B  #1,FLAG          PONI A 1 FLAG FINE RIGA
00004620: 31C8 6002   DONE     MOVE.W  A0,POINTER       AGGIORNA PUNTATORE BUFFER
00004624: 205F                 MOVE.L  (SP)+,A0         RITORNA ALLA ROUTINE INTERROTTA
00004626: 4E73                 RTE                      RIPRISTINA REGISTRO A0

                               END     PGM15_3B
```

Utilizzando la doppia bufferizzazione, le operazioni di elaborazione e di input/output possano avvenire contemporaneamente senza che la CPU debba attendere che la stampante sia pronta.

Buffer di Lunghezza Costante

Un altro metodo è quello di utilizzare la variabile FLAG come contatore di buffer. Ad esempio, il programma seguente attende che siano stati inviati alla stampante 20 caratteri.

Programma 15-3c

```
00004000:             PROGRAM  EQU     $4000
00004600:             INT_25   EQU     $4600
00006000:             DATA     EQU     $6000
                      ;
0003FF40:             PIADDA   EQU     $3FF40           REGISTRO DIREZIONE DATI A
000000FF:             DLOUT    EQU     $FF              TUTTE LE LINEE SONO USCITE
0003FF40:             PIADA    EQU     $3FF40           REGISTRO DATI A
0003FF44:             PIACA    EQU     $3FF44           REGISTRO DI CONTROLLO A
00000005:             PIA_EN   EQU     $05              ABILITAZIONE INTERRUPT
00002000:             IMSK0    EQU     $2000            SUPERVISORE/INTERRUPT LIV. 0
0000000D:             CR       EQU     $0D              RITORNO CARRELLO
                      ;
                               ORG     DATA

00006000:             FLAG     DS.B    1                CONTATORE DEL BUFFER
00006001:                      DS.B    1
00006002:             POINTER  DS.W    1                PUNTATORE FINE BUFFER + 1
00006004:             BUFFER   DS.B    80               BUFFER DI INPUT
                      ;
                               ORG     PROGRAM

00004000: 4238 6000   PGM15_3C CLR.B   FLAG             AZZERA CONTATORE BUFFER
00004004: 31FC 6004
00004008: 6002                 MOVE.W  #BUFFER,POINTER  INIZIALIZZA PUNTATORE
0000400A: 4239 0003
0000400E: FF44                 CLR.B   PIACA            IND. REG. DIREZIONE DATI
                      ;
00004010: 13FC 00FF
00004014: 0003 FF40            MOVE.B  #DLOUT,PIADDA    TUTTE LE LINEE SONO USCITE
00004018: 13FC 0005
0000401C: 0003 FF44            MOVE.B  #PIA_EN,PIACA    ABILITA INTERRUP PIA STAMPANTE
00004020: 46FC 2000            MOVE    #IMSK0,SR        ABILITA TUTTI GLI INTERRUPT
00004024: 0C38 0014
00004028: 6000        WTEOL    CMP.B   #20,FLAG         GIA' INVIATI 20 CARATTERI?
0000402A: 66F8                 BNE     WTEOL            NO, ATTENDI
```

```
0000402C: 4E75              ;          RTS
                            ;
                            ;    *     ROUTINE DI SERVIZIO DEGLI INTERRUPT
                            ;
                                       ORG      INT_25

00004600: 2F08                         MOVE.L   A0,-(SP)        METTI A0 SULLO STACK SUPERVISORE
00004602: 4A39 0003
00004606: FF40                         TST.B    PIADA           AZZERA INTERRUPT STAMPANTE
00004608: 3078 6002                    MOVE.W   POINTER,A0      PRENDI PUNT.NUOVO INGRESSO
0000460C: 13D8 0003
00004610: FF40                         MOVE.B   (A0)+,PIADA     INVIA NUOVO CAR. A STAMPANTE
00004612: 0C28 000D
00004616: FFFF                         CMPI.B   #CR,-1(A0)      L'ULTIMO CAR. ERA UN RITORNO CARRELLO?
00004618: 6604                         BNE.S    DONE            NO,RITORNA
0000461A: 5278 6000                    ADDQ     #1,FLAG         INCREMENTA CONTATORE BUFFER
0000461E: 31C8 6002         DONE       MOVE.W   A0,POINTER      AGGIORNA PUNTATORE BUFFER
00004622: 205F                         MOVE.L   (SP)+,A0        RIPRISTINA REGISTRO A0
00004624: 4E73                         RTE                      RITORNA ALLA ROUTINE INTERROTTA

                                       END      PGM15_3C
```

15-4. Un interrupt da clock in tempo reale

Scopo: L'elaboratore attende un interrupt da un clock in tempo reale.

Clock in Tempo Reale

Un clock in tempo reale fornisce semplicemente una serie regolare di impulsi: l'intervallo fra un inpulso e l'altro serve a misurare il tempo. Contando gli interrupt provenienti da un clock, si può ottenere qualsiasi multiplo dell'intervallo fondamentale. Un clock in tempo reale si può realizzare dividendo il clock della CPU, tramite un timer, come il 6840 oppure quello incluso nel dispositivo ausiliario multifunzione 6846 o, in alternativa, utilizzando sorgenti esterne, come la frequenza di una linea di corrente alternata.

Scelta della frequenza

Bisogna valutare i vantaggi e gli svantaggi connessi alla scelta di una particolare frequenza. Una frequenza elevata (poniamo10 kHz) permette di disporre di un'ampia gamma di intervalli di tempo, tutti molto precisi. D'altra parte, questo rende notevolmente complesso il conteggio degli interrupt provenienti dal clock. **La scelta della frequenza dipende dal grado di precisione e di sincronizzazione necessari alla vostra applicazione. Naturalmente, un clock, come abbiamo detto, può essere realizzato, almeno parzialmente, tramite hardware con un contatore che si assuma l'incarico di contare gli impulsi, limitandosi ad interrompere il processore solo occasionalmente. Il programma si limiterà a leggere il contatore per misurare il tempo con precisione.**

Il problema è quello di sincronizzare le varie operazioni con il clock. Chiaramente, se la CPU inizia le misurazioni in un punto qualunque del ciclo del clock, invece che all'inizio, non si riuscirà ad ottenere un elevato grado di precisione. **Questi sono alcuni dei modi in cui sincronizzare le diverse operazioni:**

1. **Avviare il clock e la CPU contemporaneamente**, mediante un interrupt di startup o RESET.
2. **Consentire alla CPU di avviare e bloccare il clock, sotto controllo del programma.**

3. **Usare un clock con una frequenza elevata,** in modo che un errore inferiore ad un ciclo di clock risulti trascurabile.
4. **Allineare il clock** (apettando un fronte o un interrupt) **prima di iniziare le misurazioni.**

Priorità della frequenza

Un interrupt proveniente da un clock in tempo reale deve avere una priorità molto elevata, in quanto anche un piccolo ritardo nel servire l'interrupt altera la precisione degli intervalli di tempo. **Solitamente ad un interrupt da clock in tempo reale viene assegnato il livello più alto di priorità, subito dopo quello riservato ad un interrupt causato da una mancanza di corrente.** La routine di servizio di un interrupt da clock è, in genere, molto breve, in modo tale che non interferisca con le altre attività della CPU.

I programmi seguenti presuppongono la presenza di un clock collegato ad una linea di interrupt del PIA. Un interrupt si verificherà ad ogni ciclo di clock.

15-4a. Attesa di un Clock in Tempo Reale

Programma 15-4a

```
00004000:                  PROGRAM  EQU     $4000
00004600:                  INT_26   EQU     $4600
00006000:                  DATA     EQU     $6000
                           ;
0003FF40:                  TPIADA   EQU     $3FF40          REG. DATI A PER IL PIA DEL TIMER
0003FF44:                  TPIACA   EQU     $3FF44          REG. DI CONTROLLO A PIA TIMER
00000005:                  PIA_EN   EQU     $05             ABILITA INTERRUPT DEL PIA
00002000:                  IMSK0    EQU     $2000           SUPERVISORE/INTERRUPT LIV. 0
                           ;
                                    ORG     DATA

00006000:                  COUNTER  DS.B    1               CONTATORE TIMER
                           ;
                                    ORG     PROGRAM

00004000: 4238 6000        PGM15_4A CLR.B   COUNTER         AZZERA CONTATORE TIMER
00004004: 13FC 0005
00004008: 0003 FF44                 MOVE.B  #PIA_EN,TPIACA  ABILITA INTERRUPT PIA TIMER
0000400C: 46FC 2000                 MOVE    #IMSK0,SR       ABILITA TUTTI GLI INTERRUPT
00004010: 4A38 6000        TWAIT    TST.B   COUNTER         IL CONTATORE E' STATO INCREMENTATO?
00004014: 67FA                      BEQ     TWAIT           NO,ATTENDI
                           ;
00004016: 4E75                      RTS
                           ;
                           ;   *    ROUTINE DI SERVIZIO INTERRUPT DEL TIMER
                           ;
                                    ORG     INT_26

00004600: 4A39 0003
00004604: FF40                      TST.B   TPIADA          AZZERA INTERRUPT TIMER
00004606: 5238 6000                 ADDQ.B  #1,COUNTER      INCREMENTA CONTATORE DEL TIMER
0000460A: 4E73                      RTE                     RITORNA ALLA ROUTINE INTERROTTA

                                    END     PGM15_4A
```

Descrizione del programma 15-4a

La variabile COUNTER, alla locazione di memoria 6000, contiene il contatore di clock.

Se il bit 1 del registro di controllo del PIA è 0 l'interrupt si verifica in corrispondenza del fronte di discesa (alto-basso) di un impulso. Se il suo valore è 1 l'interrupt si avrà sul fronte di salita (basso-alto) del clock.

La routine di servizio dell'interrupt provvede ad azzerare il bit 7 del registro di controllo, dal momento che non è necessario nessun trasferimento di dati.

La porta dati del PIA non viene utilizzata e resta disponibile per altri eventuali impieghi, a condizione che non venga azzerato accidentalmente il bit di stato corrispondente al clock in tempo reale, prima che esso sia stato riconosciuto. Questo non accade, nel caso che la porta sia utilizzata come uscita verso una periferica piuttosto semplice come un gruppo di LED, in quanto le operazioni di output non modificherebbero in alcun modo i bit di stato.

Per ampliare questa routine in modo da gestire più contatori ed ottenere una maggiore precisione, basta utilizzare un maggior numero di locazioni per il contatore di clock e un differente tipo di test nel programma principale.

15-4b. Attendere 10 Interrupt da Clock

Programma 15-4b

```
00004000:                PROGRAM  EQU      $4000
00004600:                INT_26   EQU      $4600
00006000:                DATA     EQU      $6000
                         ;
0003FF40:                TPIADA   EQU      $3FF40          REG. DATI A PER IL PIA DEL TIMER
0003FF44:                TPIACA   EQU      $3FF44          REG. DI CONTROLLO A PIA TIMER
00000005:                PIA_EN   EQU      $05             ABILITA INTERRUPT DEL PIA
00002000:                IMSK0    EQU      $2000           SUPERVISORE/INTERRUPT LIV. 0
0000000A:                TDELAY   EQU      10              RITARDO DEL TIMER
                         ;
                                  ORG      DATA

00006000:                COUNTER  DS.B     1               CONTATORE TIMER
                         ;
                                  ORG      PROGRAM

00004000: 4238 6000      PGM15_4B CLR.B    COUNTER         AZZERA CONTATORE TIMER
00004004: 13FC 0005
00004008: 0003 FF44               MOVE.B   #PIA_EN,TPIACA  ABILITA INTERRUPT PIA TIMER
0000400C: 46FC 2000               MOVE     #IMSK0,SR       ABILITA TUTTI GLI INTERRUPT
00004010: 103C 000A               MOVE.B   #TDELAY,D0      DURATA DEL RITARDO
00004014: B038 6000      TWAIT    CMP.B    COUNTER,D0      E' TRASCORSO L'INTERVALLO PREVISTO?
00004018: 67FA                    BEQ      TWAIT           NO,ATTENDI
                         ;
0000401A: 4E75                    RTS
                         ;
                         ;   *    ROUTINE DI SERVIZIO INTERRUPT DEL TIMER
                         ;
                                  ORG      INT_26

00004600: 4A39 0003
00004604: FF40                    TST.B    TPIADA          AZZERA INTERRUPT TIMER
00004606: 5238 6000               ADDQ.B   #1,COUNTER      INCREMENTA CONTATORE DEL TIMER
0000460A: 4E73                    RTE                      RITORNA ALLA ROUTINE INTERROTTA

                                  END      PGM15_4B
```

15-4c. Conservare il Tempo Reale

Una routine più verosimile per la gestione di un interrupt proveniente da un clock in tempo reale dovrebbe tener conto anche del tempo trascorso, servendosi di un certo numero di locazioni di memoria. Ad esempio, la routine seguente utilizza per questo scopo gli indirizzi compresi fra 6000 e 6003:

6000 - centesimi di secondo
6001 - secondi
6002 - minuti
6003 - ore

Si presuppone la presenza di un input che produca degli interrupt ad una frequenza costante di 100Hz.

Programma 15-4c

```
00004000:                   PROGRAM  EQU      $4000
00004600:                   INT_26   EQU      $4600
00006000:                   DATA     EQU      $6000
                            ;
0003FF40:                   TPIADA   EQU      $3FF40          REG. DATI A PER IL PIA DEL TIMER
0003FF44:                   TPIACA   EQU      $3FF44          REG. DI CONTROLLO A PIA TIMER
00000005:                   PIA_EN   EQU      $05             ABILITA INTERRUPT DEL PIA
00002000:                   IMSK0    EQU      $2000           SUPERVISORE/INTERRUPT LIV. 0
0000001E:                   TDELAY   EQU      30              INTERVALLO DEL TIMER
                            ;
                                     ORG      DATA

00006000:                   HUNDSEC  DS.B     1               CENTESIMI DI SECONDO
00006001:                   SECONDS  DS.B     1               SECONDI
00006002:                   MINUTES  DS.B     1               MINUTI
00006003:                   HOURS    DS.B     1               ORE
                            ;
                                     ORG      PROGRAM

00004000: 13FC 0005
00004004: 0003 FF44         PGM15_4C MOVE.B   #PIA_EN,TPIACA  ABILITA INTERRUPT PIA TIMER
00004008: 46FC 2000                  MOVE     #IMSK0,SR       ABILITA TUTTI GLI INTERRUPT
0000400C: 1038 6000                  MOVE.B   HUNDSEC,D0      PRENDI CENTESIMI DI SECONDO
00004010: 0600 001E                  ADDI.B   #TDELAY,D0      SOMMALI ALLA DURATA DEL RITARDO
00004014: 0C00 0064                  CMPI.B   #100,D0         MODULO 100
00004018: 6500 0006                  BCS      TWAIT
0000401C: 0400 0064                  SUBI.B   #100,D0
00004020: B038 6000         TWAIT    CMP.B    HUNDSEC,D0      E' TRASCORSO L'INTERVALLO PREVISTO?
00004024: 67FA                       BEQ      TWAIT           NO, ATTENDI
                            ;
00004026: 4E75                       RTS
                            ;
                            ;   *    ROUTINE DI SERVIZIO INTERRUPT DEL TIMER
                            ;
                                     ORG      INT_26

00004600: 48E7 8000                  MOVEM.L  D0,-(SP)        SALVA D0
00004604: 4A39 0003
00004608: FF40                       TST.B    TPIADA          AZZERA INTERRUPT DEL TIMER
0000460A: 5238 6000                  ADDQ.B   #1,HUNDSEC      AGGIORNA CENT. DI SECONDO
0000460E: 103C 0064                  MOVE.B   #100,D0
00004612: B038 6000                  CMP.B    HUNDSEC,D0      C'E' UN RIPORTO DAI CENTESIMI?
00004616: 6628                       BNE.S    TDONE           NO, FATTO
00004618: 4238 6000                  CLR.B    HUNDSEC         SI', AZZERA CENT. DI SECONDO
0000461C: 5238 6001                  ADDQ.B   #1,SECONDS      AGGIORNA I SECONDI
00004620: 103C 003C                  MOVE.B   #60,D0
00004624: B038 6001                  CMP.B    SECONDS,D0      C'E' UN RIPORTO DAI MINUTI?
00004628: 6616                       BNE.S    TDONE           NO, FATTO
0000462A: 4238 6001                  CLR.B    SECONDS         SI', AZZERA I SECONDI
0000462E: B038 6002                  CMP.B    MINUTES,D0      C'E' UN RIPORTO PER LE ORE?
00004632: 660C                       BNE.S    TDONE           NO, FATTO
00004634: 4238 6002                  CLR.B    MINUTES         SI', AZZERA I MINUTI
00004638: 5238 6003                  ADDQ.B   #1,HOURS        AGGIORNA LE ORE
0000463C: 5238 6002                  ADDQ.B   #1,MINUTES      AGGIORNA I MINUTI
00004640: 4CDF 0001         TDONE    MOVEM.L  (SP)+,D0        RIPRISTINA D0
00004644: 4E73                       RTE                      [illegible]

                                     END      PGM15_4C
```

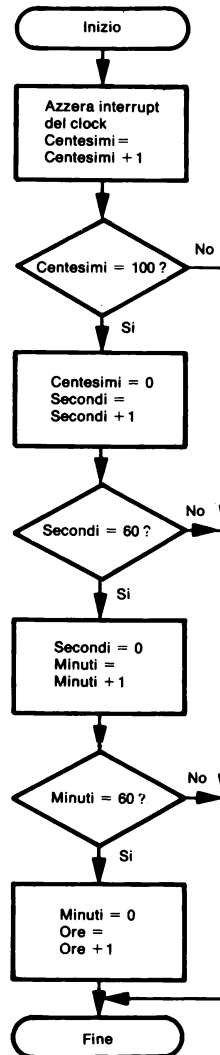
Il programma principale genera un intervallo di 300 millisecondi; l'intervallo più lungo ottenibile con questa routine è di 990 millisecondi. Quali sono le modifiche necessarie per ottenere degli intervalli di maggior durata?

È lo stesso metodo che utilizziamo quando dobbiamo far cuocere qualcosa per 20 minuti. Guardiamo l'orologio (contatore) per vedere l'ora, a questa sommiamo 20 (in modulo 60, per cui 20 minuti dopo le 6.50 corrispondono alle 7.10) per stabilire il tempo finale ed attendiamo che questo venga raggiunto. Se l'intervallo è inferiore ad un'ora, si può trascurare la lancetta delle ore ed attendere finchè quella dei minuti non ne indica dieci dopo l'ora. E questo il metodo usato dal programma. (Se il vostro orologio non ha le lancette, aspettate finchè le cifre dei minuti non indicano10).

Modificare il programma in modo da generare un ritardo di 20 minuti (necessario, ad es., per un forno a microonde controllato da un microprocessore).

Naturalmente, il programma potrebbe svolgere altre funzioni, effettuando solo controlli occasionali del tempo trascorso. In che modo potreste ottenere un ritardo di sette secondi oppure di tre minuti? Molte applicazioni non richiedono che intervalli di lunga durata siano particolarmente precisi; ad esempio, l'operatore di un forno a microonde non si preoccupa del fatto che un intervallo dell'ordine di minuti possa durare qualche secondo in più o in meno.

In alcuni casi il tempo viene memorizzato sotto forma di cifre BCD o di caratteri ASCII. Sareste in grado di modificare la precedente routine di interrupt, inserendo queste opzioni?

Diagramma di Flusso 15-4a

Qualora il PIA del clock generi degli interrupt di livello 2, essi saranno gestiti attraverso l'autovettore di livello 2, posto all'indirizzo 68. Vediamo cosa accade se la maschera di priorità è posta al livello 0 e si verificano due interrupt simultanei provenienti dal PIA del clock e da quello della stampante dell'esempio 15-3. Dal momento che il PIA della stampante genera un interrupt di livello 1, il PIA del clock viene servito per primo, mentre quello della stampante è inibito fino a quando la maschera delle priorità non ritorna a zero. Se per primo si verifica l'interrupt della stampante ed è già cominciato il relativo servizio, questo viene interrotto dall'interrupt del clock. Quando è terminata la routine che serve il clock il controllo ritorna alla routine della stampante, nello stesso punto in cui era stata interrotta.

Un Clock ad Alta Frequenza

Anche un clock in tempo reale ad alta frequenza può essere gestito senza che il processore debba intervenire troppo spesso. Il metodo adottato comunemente prevede che il clock incrementi un insieme di contatori, i quali generano, poi, degli interrupt ad una frequenza molto più bassa. Ad esempio, una frequenza di input pari ad 1 MHz viene fatta passare attraverso 3 contatori decimali e l'uscita dell'ultimo viene collegata al PIA. Il PIA riceverà un impulso di clock ogni 1000 impulsi d'ingresso (cioè, quando si verifica un overflow dei 3 contatori decimali). Il processore può, se necessario, determinare il tempo con una precisione superiore ad 1 ms. leggendo i contatori, che contengono le cifre meno significative. Di solito, viene impiegato dell'hardware aggiuntivo (contatori e porte di input) per non sovraccaricare eccessivamente la CPU. Anche in questo caso è necessaria una valutazione dei pro e dei contro: varrà la pena di utilizzare dell'hardware aggiuntivo solo se l'applicazione richiede un elevato grado di precisione.

15-5. Un interrupt da telescrivente

15-5a. Routine per un Interrupt ACIA

Scopo: Il programma principale azzera il flag rappresentato dalla variabile FLAG, alla locazione 6000, ed attende un interrupt da un ACIA 6850. La routine per il servizio dell'interrupt pone FLAG a 1 e mette il dato provenienete dall'ACIA nella variabile CHAR, alla locazione di memoria 6001. I caratteri hanno una lunghezza di 7 bit, parità dispari e 2 bit di stop.

Programma 15-5a

```
00004000:                PROGRAM  EQU     $4000
00004600:                INT_25   EQU     $4600
00006000:                DATA     EQU     $6000
                         ;
0003FF01:                ACIACR   EQU     $3FF01           REGISTRO DI CONTROLLO DELL'ACIA
000EFF03:                ACIADR   EQU     $EFF03           REGISTRO DATI DELL'ACIA
000000C5:                AMODE    EQU     $C5              MODO OPERATIVO DELL'ACIA
00002000:                MRESET   EQU     $2000            MASTER RESET DELL'ACIA
00002000:                IMSK0    EQU     $2000            SUPERVISORE/INTERRUPT LIV. 0
                         ;
                                  ORG     DATA
00006000:                FLAG     DS.B    1                FLAG DATO ACCETTATO
00006001:                CHAR     DS.B    1                CARATTERE DALLA TELESCRIVENTE
                         ;
                                  ORG     PROGRAM
00004000: 4238 6000      PGM15_5A CLR.B   FLAG             AZZERA FLAG DATO ACCETTATO
00004004: 13FC 0000
00004008: 0003 FF01               MOVE.B  #MRESET,ACIACR   MASTER RESET DELL'ACIA
0000400C: 13FC 00C5
00004010: 0003 FF01               MOVE.B  #AMODE,ACIACR    ABILITA INTERRUPT ACIA/SEL.MODO
00004014: 46FC 2000               MOVE    #IMSK0,SR        ABILITA TUTTI GLI INTERRUPT
00004018: 4A38 6000      WAIT     TST.B   FLAG             C'E' UN DATO DALL'ACIA?
0000401C: 67FA                    BEQ     WAIT             NO,ATTENDI
                         ;
0000401E: 4E75                    RTS
                         ;
                         ;   *    ROUTINE DI SERVIZIO DEGLI INTERRUPT
                         ;
                                  ORG     INT_25
00004600: 11FC 0001
00004604: 6000                    MOVE.B  #1,FLAG          PONI A 1 FLAG DATO ACCETTATO
00004606: 11F9 000E
0000460A: FF03 6001               MOVE.B  ACIADR,CHAR      SALVA L'INPUT DALLA TTY
0000460E: 4E73                    RTE                      RITORNA ALLA ROUTINE INTERROTTA

                                  END     PGM15_5A
```

Dal momento che l'ACIA 6850 non ha un ingresso di RESET, è necessario un Master Reset (ponendo a 1 contemporaneamente il bit 0 ed il bit 1 del registro di controllo), prima di inizializzare l'ACIA.

Successivamente, inizializzeremo il registro di controllo nel modo seguente:

Bit 7 = 1 per abilitare l'interrupt del ricevitore
Bit 6 = 1 e Bit 5 = 0 per disabilitare l'interrupt del trasmettitore
Bit 4 = 0, Bit 3 = 0 e Bit 2 = 1 per selezionare dati a 7 bit, con parità dispari e 2 bit di Stop
Bit 1 = 0 e Bit 0 = 1 per selezionare il modo clock 16 (per una velocità di 110 Baud bisogna fornire un clock con una frequenza di 1760 Hz).

Per stabilire quale ACIA ha generato l'interrupt, il programma deve esaminare il bit relativo alla richiesta d'interrupt (bit 7 del registro di stato) di tutti gli ACIA presenti. Mentre, per distinguere fra gli interrupt del ricevitore e quelli del trasmettitore, bisogna esaminare il bit che indica se il registro ricezione dati è pieno (bit 0 del registro di stato). Il bit di richiesta d'interrupt dell'ACIA viene azzerato sia leggendo il registro ricezione dati, che scrivendo nel registro trasmissione dati.

15-5b. Un Interrupt dal Bit di Start del PIA

Un dato proveniente da una telescrivente può essere ricevuto anche tramite un PIA. In tal caso, la linea d'ingresso seriale, prove-

niente dalla telescrivente, è collegata sia al bit 7 del registro dati che alla linea di controllo 1 del PIA.

Scopo: Il programma principale azzera un flag rappresentato dalla variabile FLAG, alla locazione di memoria 6000, ed attende un interrupt dalla telescrivente. La routine per il servizio dell'interrupt pone FLAG a 1 e mette il dato della telescrivente nella variabile CHAR, alla locazione 6001.

Programma 15-5b

```
00004000:                 PROGRAM  EQU     $4000
00004600:                 INT_25   EQU     $4600
00006000:                 DATA     EQU     $6000
00004800:                 TTYRCV   EQU     $4800
                          ;
0003FF40:                 PIADDA   EQU     $3FF40           REGISTRO DIREZIONE DATI A
00000000:                 DATIN    EQU     $0               LE LINEE DATI SONO TUTTE INGRESSI
0003FF40:                 PIADA    EQU     $3FF40           REGISTRO DATI A
0003FF44:                 PIACA    EQU     $3FF44           REGISTRO DI CONTROLLO A
00000005:                 PIA_EN   EQU     $05              ABILITAZIONE INTERRUPT DEL PIA
00000004:                 PIA_DIS  EQU     $04              DISABILITA INTERRUPT DEL PIA
00002000:                 IMSK0    EQU     $2000            SUPERVISORE/INTERRUPT LIV. 0
                          ;
                                   ORG     DATA

00006000:                 FLAG     DS.B    1                FLAG DATO ACCETTATO
00006001:                 CHAR     DS.B    1                CARATTERE DI INPUT DALLA TTY
                          ;
                                   ORG     PROGRAM

00004000: 4238 6000       PGM15_5B CLR.B   FLAG             AZZERA FLAG DATO ACCETTATO
00004004: 4239 0003
00004008: FF44                     CLR.B   PIACA            IND. REG. DIREZIONE DATI
                          ;
0000400A: 13FC 0000
0000400E: 0003 FF40                MOVE.B  #DATIN,PIADDA    TUTTE LE LINEE SONO USCITE
00004012: 13FC 0005
00004016: 0003 FF44                MOVE.B  #PIA_EN,PIACA    ABILITA INTERRUP PIA TTY
0000401A: 46FC 2000                MOVE    #IMSK0,SR        ABILITA TUTTI GLI INTERRUPT
0000401E: 4A38 6000       WAIT     TST.B   FLAG             E' STATO RICEVUTO IL BIT DI START?
00004022: 67FA                     BEQ     WAIT             NO,ATTENDI
00004024: 4EB8 4800                JSR     TTYRCV           SI', PRENDI UN DATO DALLA TTY
00004028: 11C0 6001                MOVE.B  D0,CHAR          SALVA IL CAR. DI INPUT DALLA TTY
                          ;
0000402C: 4E75                     RTS
                          ;
                          ;   *    ROUTINE DI SERVIZIO DEGLI INTERRUPT
                          ;
                                   ORG     INT_25

00004600: 11FC 0001
00004604: 6000                     MOVE.B  #1,FLAG          PONI A 1 FLAG DATO ACCETTATO
00004606: 4A39 0003
0000460A: FF40                     TST.B   PIADA            AZZERA INTERRUPT BIT DI START
0000460C: 13FC 0004
00004610: 0003 FF44                MOVE.B  #PIA_DIS,PIACA   DISABILITA INTERRUPT
00004614: 4E73                     RTE                      RITORNA ALLA ROUTINE INTERROTTA

                                   END     PGM15_5B
```

La subroutine TTYRCV, richiamata dal Programma 15-5b, è simile alla routine per la ricezione da telescrivente mostrata nell'Esempio 9 del Capitolo 13, tranne per il fatto di lasciare un dato nel registro dati D0. Il fronte usato per provocare l'interrupt, in questo caso, è molto importante. La transizione che deve causare l'interrupt è quella dallo stato normale '1' (Mark) allo stato '0' (Spazio), poichè è questo tipo di transizione che indica anche l'inizio della trasmissione. Una transizione da '0' a '1' non si verifica finchè non viene ricevuto un bit del dato diverso da zero.

La routine di servizio disabilita l'interrupt del PIA poichè, altrimenti, ogni transizione da '1' a '0' durante l'invio del carattere provocherà un interrupt. Osservate come la lettura dei bit del dato azzeri tutti i flag di stato messi a 1 dalle transizioni che devono essere ignorate. Naturalmente, il programma riabilita l'interrupt del PIA

(ponendo a 1 il bit 0 del registro di controllo), per consentire la ricezione del dato successivo, ma questo lo fa solamente dopo aver letto il carattere presente nel registro dati.

15-6. Una Chiamata in Modo Supervisore

Scopo: Permettere a dei programmi in modo Utente di accedere a routine di utility in modo Supervisore.

Nella progettazione di sistemi che includono un monitor o un sistema operativo è buona norma trasformare in routine di utility delle sequenze di istruzioni utilizzate molto frequentemente. Queste routine possono svolgere delle funzioni piuttosto semplici, come stabilire l'ora, oppure altre molto più complesse, come gestire la memoria in un sistema multiutente o input/output logici in sistemi che utilizzano dischi come memorie di massa. L'architettura dell'MC68000 impedisce ai programmi applicativi in modo Utente di eseguire certe istruzioni privilegiate, riservate al modo Supervisore. In sistemi futuri che consentano una gestione della memoria, i programmi in modo Utente utilizzeranno probabilmente solo un'area di memoria posta entro determinati limiti di indirizzamento.

Quando i programmi in modo Utente devono comunicare con un monitor o un sistema operativo nel modo Supervisore, bisogna far ricorso alle istruzioni TRAP, che sono in grado di provocare un'Exception del processore, gestita in modo analogo agli interrupt. I Programmi 15-6a e 15-6b mostrano due fra gli usi più frequenti dell'istruzione TRAP.

Programma 15-6a

```
00004000:                 PROGRAM  EQU      $4000
00004400:                 TTYIN    EQU      $4400
00004500:                 PRINT    EQU      $4500
00004600:                 TRAP1    EQU      $4600
00005100:                 USTACK   EQU      $5100
00006000:                 DATA     EQU      $6000
                          ;
                                   ORG      $84             VETTORE TRAP 1
00000084:                          DC.L     TRAP1
                                   ORG      DATA
00006000:                 BUFFER   DS.B     80              BUFFER DI INPUT/OUTPUT
                          ;
                                   ORG      PROGRAM
                          ;
                          ;   *    PROGRAMMA IN MODO UTENTE
                          ;
00004000: 3C7C 6000       PGM15_6A MOVE.W   #BUFFER,A6      PUNT. AL BUFFER DI INPUT/OUTPUT
00004004: 4E41                     TRAP     #1              CHIAMATA AL MONITOR
00004006:                          DC.W     1               PER LEGGERE UNA RIGA DALLA TTY
00004008: 4E41                     TRAP     #1              CHIAMATA AL MONITOR
0000400A:                          DC.W     2               PER SCRIVERE UNA RIGA SULLA STAMPANTE
0000400C: 4E75                     RTS
                          ;
                          ;   *    ROUTINE GESTIONE TRAP1
                          ;
                                   ORG      TRAP1
00004600: 48E7 FFFE                MOVEM.L  D0-D7/A0-A6,-(SP) SALVA TUTTI I REG. UTENTE
00004604: 2A6F 003E                MOVE.L   60+2(SP),A5     INDIRIZZO DI RITORNO
00004608: 4BED 0002                LEA      2(A5),A5        IND. DELL'ISTRUZIONE DOPO LA TRAP
0000460C: 2F4D 003E                MOVE.L   A5,60+2(SP)     AGGIORNA IL VALORE DELLO STACK
00004610: 0C6D 0001
00004614: FFFE                     CMP.W    #1,-2(A5)       CHIAMATA DI LETTURA?
00004616: 6606                     BNE.S    PRINTER         NO, CHIAMATA DI STAMPA
00004618: 4EB8 4400                JSR      TTYIN           LEGGI UNA LINEA DALLA TTY
0000461C: 6004                     BRA.S    DONE
0000461E: 4EB8 4500       PRINTER  JSR      PRINT           INVIA UNA LINEA ALLA STAMPANTE
00004622: 4CDF 7FFF       DONE     MOVEM.L  (SP)+,D0-D7/A0-A6 RIPRISTINA REGISTRI UTENTE
00004626: 4E73                     RTE                      RITORNA AL PROGRAMMA UTENTE

                                   END      PGM15_6A
```

Descrizione del programma 15-6a

Ciascuno dei due modi operativi del processore dispone del proprio puntatore allo stack (registro indirizzi A7). Quando viene eseguito un reset, il registro indirizzi A7 è utilizzato come puntatore allo stack del modo Supervisore, finchè non sarà selezionato il modo Utente mediante l'azzeramento del bit S del registro di stato.

Il Programma 15-6a mostra una tipica sequenza di istruzioni necessaria per scrivere e leggere dati da una telescrivente, utilizzando un monitor come il MACSBUG della Motorola. L'istruzione TRAP #1 richiama una funzione in modo Supervisore. In questo esempio, il registro indirizzi A6 contiene un parametro di input per la funzione e punta al buffer di input/output della TTY. Un secondo parametro è presente nella word immediatamente successiva all'istruzione TRAP: indica se la richiesta riguarda una funzione di ingresso oppure una di uscita. Una descrizione dettagliata delle modalità per il trasferimento dei parametri è contenuta nel Capitolo 10.

Come abbiamo detto all'inizio di questo capitolo, in seguito all'Exception dovuta all'istruzione TRAP sia il contatore di programma che il registro di stato vengono messi sullo stack Supervisore. Il numero di TRAP (in questo caso 1), è usato per individuare il vettore corrispondente, analogamente a quanto accade con un vettore di interrupt. Dal momento che i vettori di TRAP iniziano a partire dall'indirizzo $80, il vettore relativo a TRAP #1 si trova a:

$$\$80 + 1 * 4 = \$84$$

La long word alla locazione $84 contiene l'indirizzo di partenza della routine che gestisce TRAP #1, corrispondente alla locazione $4600.

Dato che durante la gestione dell'Exception sono salvati automaticamente solo il registro di stato ed il contatore di programma, la nostra routine deve salvare qualunque altro registro intenda utilizzare, provvedendo a ripristinarli prima di ritornare all'istruzione successiva all'Exception. Nel caso che il controllo non sia immediatamente restituito al programma che ha provocato l'Exception, sarà opportuno salvare anche il registro dello stack Utente, con l'istruzione MOVE USP,An. Al momento di ritornare al programma principale, l'istruzione MOVE An,USP provvede a ripristinare il puntatore dello stack Utente. Sono entrambe istruzioni privilegiate e sono indispensabili nei sistemi multitasking.

Una volta completata la prevista routine di risposta il controllo ritorna all'istruzione successiva a quella che ha causato l'Exception. Questo è possibile grazie all'istruzione RTE, che ripristina anche il registro di stato ed il contatore di programma, già salvati in precedenza sullo stack Supervisore. Dal momento che RTE agisce su quella parte del registro di stato riservata al modo Supervisore, si tratterà di un'istruzione privilegiata.

Il programma 15-6b è una variante di quello precedente. Sono utilizzate due diverse istruzioni TRAP e, quindi, anche due routine di gestione. Normalmente ricorriamo alle istruzioni TRAP quando ci troviamo nel modo Utente e vogliamo comunicare con delle

funzioni in modo Supervisore, ma, in realtà, è possibile utilizzarle anche in modo Supervisore.

Programma 15-6b

```
00004000:                     PROGRAM  EQU      $4000
00004400:                     TTYIN    EQU      $4400
00004500:                     PRINT    EQU      $4500
00004600:                     TRAPHDLR EQU      $4600
00005100:                     USTACK   EQU      $5100
00006000:                     DATA     EQU      $6000
                              ;
                                       ORG      $84              VETTORE TRAP 1/2

00000084:                              DC.L     TRAP1
00000088:                              DC.L     TRAP2
                              ;
                                       ORG      DATA

00006000:                     BUFFER   DS.B     80               BUFFER DI INPUT/OUTPUT
                              ;
                                       ORG      PROGRAM
                              ;
                              ;   *    PROGRAMMA IN MODO UTENTE
                              ;

00004000: 3E7C 5100           PGM15_6B MOVEA.W  #USTACK,A7       INIZIALIZZA STACK UTENTE
00004004: 3C7C 6000                    MOVE.W   #BUFFER,A6       PUNT. AL BUFFER DI INPUT/OUTPUT
00004008: 4E41                         TRAP     #1               CHIAMATA AL MONITOR PER LETTURA
0000400A: 4E41                         TRAP     #1               CHIAMATA AL MONITOR PER SCRITTURA
0000400C: 4E75                         RTS
                              ;
                              ;   *    ROUTINE GESTIONE TRAP 1 E 2
                              ;
                                       ORG      TRAPHDLR

00004600: 48E7 FFFE           TRAP1    MOVEM.L  D0-D7/A0-A6,-(SP) SALVA TUTTI I REG. UTENTE
00004604: 4EB8 4400                    JSR      TTYIN            LEGGI UNA LINEA DALLA TTY
00004608: 6008                         BRA.S    RETURN
0000460A: 48E7 FFFE           TRAP2    MOVEM.L  D0-D7/A0-A6,-(SP) SALVA TUTTI I REG. UTENTE
0000460E: 4EB8 4500                    JSR      PRINT            INVIA UNA LINEA ALLA STAMPANTE
00004612: 4CDF 7FFF           RETURN   MOVEM.L  (SP)+,D0-D7/A0-A6 RIPRISTINA REGISTRI UTENTE
00004616: 4E73                         RTE                       RITORNA AL PROGRAMMA UTENTE

                                       END      PGM15_6B
```

15-7. Passaggio al modo utente

Scopo: Trasferire il controllo a programmi in modo Utente

Programma 15-7

```
00004800:                     RESET    EQU      $4800
00005100:                     STACK    EQU      $5100
00005300:                     USTACK   EQU      $5300
00000000:                     USER     EQU      $0               MODO UTENTE/LIV. DI PRIORITA' 0
00004000:                     USERPGM  EQU      $4000            PROGRAMMA UTENTE
                              ;
                                       ORG      RESET

00004800: 307C 5300           PGM15_7  MOVEA.W  #USTACK,A0       INDIRIZZO STACK UTENTE
00004804: 4E60                         MOVE.L   A0,USP           INIZIALIZZA STACK UTENTE
00004806: 46FC 0000                    MOVE.W   #USER,SR         SELEZIONA MODO UTENTE
0000480A: 4EF8 4000                    JMP      USERPGM          VAI AL PROGRAMMA UTENTE

                                       END      PGM15_7
```

Azzeramento del flag Supervisore (bit S)

Come abbiamo già ricordato, l'MC68000 viene inizializzato per operare in modo Supervisore; volendo selezionare il modo Utente sarà necessario azzerare il flag Supervisore (bit S) del registro di stato. Questo si può ottenere con una qualsiasi delle istruzioni che agiscono sul flag Supervisore, come MOVE a SR, ANDI con SR, EORI con SR o RTE. Le istruzioni MOVE, ANDI ed EORI

agiscono soltanto sul registro di stato e l'istruzione successiva viene eseguita nel modo Utente. Con l'istruzione RTE è possibile passare al modo Utente indicando anche un indirizzo specifico.

ROUTINE DI SERVIZIO PIÙ GENERALI

In un sistema governato mediante interrupt le routine di servizio hanno carattere più generale e devono assolvere i seguenti compiti:

1. **Salvare tutti i dati necessari sullo stack, in modo che il programma che ha subito l'interruzione possa riprendere correttamente.** Durante la risposta ad un interrupt, l'MC68000 salva, automaticamente, soltanto il contatore di programma ed il registro di stato, mettendoli sullo stack Supervisore. Perciò, le vostre routine devono provvedere a salvare qualunque altro registro di cui intendano fare uso.
2. **Ripristinare registri e dati prima di eseguire l'istruzione RTE** e restituire il controllo al programma che aveva subito l'interruzione.
3. **Effettuare il polling di tutti i dispositivi associati ad un determinato interrupt** quando questo può essere attribuito a più di una periferica. Si tratta, in genere, di dispositivi che utilizzano l'autovectoring.
4. **Abilitare e disabilitare gli interrupt in modo corretto.** La CPU disabilita automaticamente gli interrupt con una priorità uguale o inferiore a quella dell'interrupt che è stato appena riconosciuto.

BIBLIOGRAFIA

1. A. Osborne. *An Introduction Microcomputer Volume 1 - Basic Concepts*. Berkeley: Osborne/McGraw-Hill, 1980, Chapter 5.
2. R.L. Baldridge. "Interrupts Add Power, Complexity to Microcomputer Software Design," EDN, August 5, 1977, pp. 67-73.
3. R. Morris. "6800 Routine Supervises Service Requests," *EDN*, October 5, 1979, pp. 73-81.
4. I.P. Breikss "Nonmaskable Interrupt Saves Processor Register Contents," *Electronics*, July 21, 1977, p. 104.
5. A. Osborne. *An Introduction to Microcomputers: Volume 2 - Some Real Microprocessors*. Berkeley: Osborne/McGraw-Hill, 1980, pp. 9-71 through 9-77.
6. R. Grappel. "Technique Avoids Interrupt Dangers," *EDN*, May 5, 1979, p. 88.

7. G. Horner. "Online Control of a Laboratory Instruments by a Timesharing Computer, *Computer Design*, February 1980, pp.90-106.
8. Per un'ulteriore trattazione ed alcuni esempi reali di progettazione di sistemi con interrupt, si consigliano i testi seguenti:
S.C. Baunach. "An Example of an MC68000-based GPIB Interface," *EDN*, September 20. 1977, pp. 125-28.
L.E. Cannon and P.S. Kreager. "Using a Microprocessor: a Real-Life Application, Part 2 - Software," *Computer Design*, October 1975, pp. 81-89.
D. Fullager et al. "Interfacing Data Converters and Microprocessors," *Electronics*, December 8, 1976, pp. 81-9.
S.A. Hill "Multiprocess Control Interface Makes Remote MP Command Possible," *EDN*, February 5, 1976, pp. 87-9.
Holderby. "Designing a Microprocessor-based Terminal for Factory Data Collection," *Computer Design*, March 1977, pp. 81-8.
A. Lange. "OPTACON Interface permits the Blind to 'Read' Digital Instruments," *EDN* February 5, 1976, pp. 84-6.
J.D. Logan and P.S. Kreager. "Using sa Microprocessor: a Real-Life Application, Part 1 - Hardware," *Computer Design*, September 1975, pp. 69-77.
A. Moore and M. Eidson. "Printer Control", Note Applicative disponibile presso la Motorola Semiconductor Products, Phoenix, Ariz.
M.C. Mulder and P.P. Fasang. "A Microprocessor Controlled Substation Alarm Logger," IECI '78 Proceedings - Industrial Applications of Microprocessors, March 20-22, 1978, pp. 2-6.
P.J. Zsombar-Murray et al. "Microprocessor Based Frequency Response Analyzer," IECI '78 Proceedings - Industrial Applications of Microprocessors, March 20-22, 1978, pp. 36-44.

I resoconti dei Meeting Annuali del gruppo di Elettronica Industriale e Strumenti di Controllo della IEEE su "Applicazioni Industriali dei Microprocessori" contengono molti articoli interessanti. I volumi (a partire dal 1975) sono disponibili presso l'IEEE Service Centre, 445 Hoes Lane, Pineataway, N.J. 088554.

SEZIONE IV

LO SVILUPPO DEL SOFTWARE

Nei capitoli precedenti abbiamo descritto la realizzazione di brevi programmi in linguaggio assembly. Naturalmente, questa è solo una piccola parte, anche se importante, dello sviluppo del software. Per un principiante scrivere dei programmi in linguaggio assembly è un compito gravoso, ma presto diventa una cosa piuttosto semplice. A questo punto, dovreste aver già acquisito una certa familiarità con i metodi di programmazione in linguaggio assembly dell'MC68000. **Nei prossimi sei capitoli vi spiegheremo in che modo esprimere determinate funzioni sotto forma di programmi e come combinare fra loro dei brevi programmi per ottenere un sistema in grado di funzionare.**

LE FASI DELLO SVILUPPO DEL SOFTWARE

Il processo di sviluppo del software si compone di varie fasi, come appare dal diagramma di flusso della Figura IV-1. Esse sono:

- **Definizione del problema**
- **Progettazione del programma**
- **Codifica**
- **Debugging (verifica)**
- **Collaudo**
- **Documentazione**
- **Manutenzione e riprogettazione**

Ciascuna di queste fasi riveste un ruolo importante nella realizzazione di un'applicazione. La codifica, ovvero la scrittura dei programmi in una forma comprensibile ad un elaboratore, è solo una delle fasi di un lungo processo.

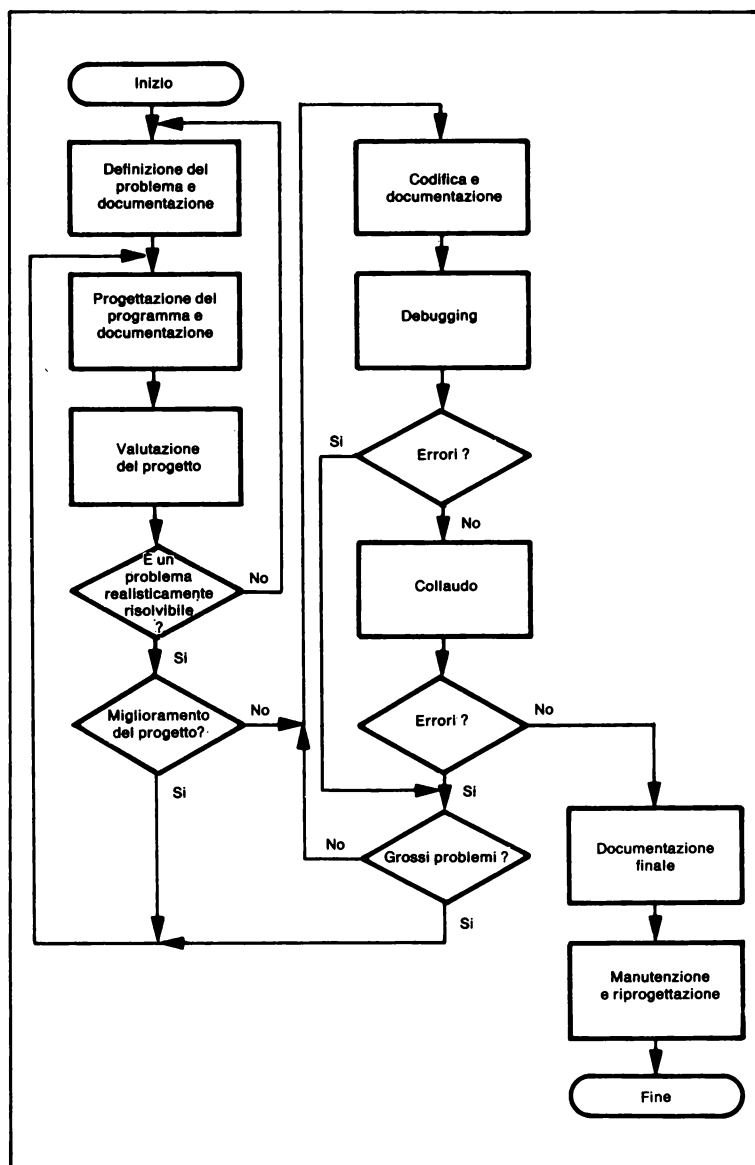



Figura IV-1. Diagramma di flusso dello Sviluppo del Software.

IMPORTANZA RELATIVA DELLA CODIFICA

La codifica rappresenta la fase più semplice sia da definire che da realizzare. Le regole per scrivere dei programmi destinati ad un elaboratore sono facili; variano leggermente da un computer all'altro, ma le tecniche fondamentali rimangono le stesse. Sono pochi i

progetti software che presentano dei problemi dovuti alla codifica, la quale, inoltre, occupa una minima parte del tempo necessario all'intero processo di sviluppo. Si ritiene che un programmatore riesca a scrivere da una a dieci istruzioni al giorno, completamente documentate e corrette. Chiaramente, la semplice codifica di un numero di istruzioni oscillante fra uno e dieci raramente richiede un'intera giornata di lavoro. Nella maggior parte dei casi, la codifica occupa meno del 25% del tempo di un programmatore.

VALUTAZIONE DEI PROGRESSI COMPIUTI NELLE ALTRE FASI

È abbastanza difficile riuscire a valutare i progressi compiuti nelle altre fasi dello sviluppo. Possiamo dire di aver scritto metà di un programma, ma difficilmente possiamo affermare di aver eliminato la metà degli errori o di aver definito la metà di un problema. È difficile prevedere con esattezza la durata di certe fasi come quelle della progettazione, del debugging e del collaudo. A volte passano dei giorni o delle settimane senza dei sostanziali progressi. Inoltre, un lavoro incompleto in una fase finisce per causare dei gravi problemi in una fase successiva. Ad esempio, una definizione approssimativa di un problema ed una progettazione mediocre rendono molto complessi il debugging ed il collaudo. Risparmiando del tempo in una fase del processo di sviluppo, si rischia di doverne perdere molto di più nelle fasi successive.

DEFINIZIONE DELLE FASI

Definizione del problema

La definizione di un problema è la formulazione di ciò che il computer deve fare per svolgere una determinata funzione. Ad esempio, che cosa è necessario per far sì che un computer controlli uno strumento, esegua una serie di collaudi elettrici o gestisca le comunicazioni fra un controllore centrale ed uno strumento posto a distanza. Durante questa fase, bisogna stabilire le modalità e la velocità delle operazioni di input/output, l'entità e la rapidità dell'elaborazione richiesta ed i tipi di errori possibili, oltre alle modalità per la loro gestione. La definizione del problema consiste nel dare un'idea sommaria del sistema che vogliamo realizzare e nel definire i compiti da assegnare all'elaboratore.

Progettazione del Programma

La progettazione comporta la stesura delle linee generali del programma che dovrà svolgere i compiti definiti nella fase precedente. Le

varie funzioni vengono descritte in modo da poter essere facilmente convertite in un programma. **Fra le tecniche che si rivelano più utili in questa fase, ricordiamo i diagrammi di flusso, la programmazione strutturata, la programmazione modulare e la progettazione "top-down"** (lett. dall'alto in basso).

Codifica

La codifica è la scrittura di un programma in una forma che un computer sia in grado di comprendere direttamente o, comunque, di tradurre. La forma può essere il linguaggio macchina, il linguaggio assembly oppure un linguaggio di alto livello.

Debugging

La fase di debugging, detta anche di verifica, consiste nel fare in modo che un programma esegua le funzioni previste. In questa fase, ci serviamo di vari strumenti, come "breakpoint" (punti di arresto), funzioni Trace, simulatori, analizzatori logici ed emulatori interni ai circuiti. È difficile stabilire la fine della fase di debugging, dal momento che non possiamo mai sapere quando ci troviamo di fronte all'ultimo errore.

Collaudo

Il collaudo, indicato anche come convalida del programma, **consiste nell'accertarsi che un programma esegua correttamente le funzioni complessive, previste nel sistema finito.** Per valutare le prestazioni del programma, il progettista si serve di simulatori, test e tecniche statistiche. Questa fase è analoga ad un controllo della qualità dell'hardware.

Documentazione

La documentazione è la descrizione del programma in una forma comprensibile agli utenti ed al personale destinato alla manutenzione. È utile anche al progettista che voglia crearsi una biblioteca di programmi da poter utilizzare in seguito. Diagrammi di flusso, mappe di memoria e moduli sono alcuni degli strumenti usati in questa fase.

Manutenzione e Riprogettazione

La manutenzione e la riprogettazione consistono nell'assistenza, nel miglioramento e nell'ampliamento del programma. Chiaramente,

un progettista deve essere in grado di risolvere problemi di funzionamento in un apparecchiatura computerizzata. Sono indispensabili metodiche e programmi di diagnosi, oltre a vari strumenti di manutenzione. Molto spesso, inoltre, è necessario ampliare un programma o aumentarne le capacità per soddisfare nuove esigenze o realizzare nuove funzioni.

CAPITOLO 16

DEFINIZIONE DI UN PROBLEMA

Le tipiche funzioni di un microprocessore devono essere definite con estrema precisione. Ad esempio, cosa deve fare un programma per controllare una bilancia, un registratore di cassa o un generatore di segnale? Chiaramente, la semplice definizione delle singole funzioni necessarie per svolgere uno di questi compiti è già un lavoro molto lungo.

INPUT

Da dove cominciare? **Naturalmente dagli input. Innanzitutto, dobbiamo elencare tutti gli input che il computer riceverà in quel determinato tipo di applicazione.**

Questi sono alcuni esempi:

- Blocchi di dati da linee di trasmissione
- Word indicanti lo stato dalle periferiche
- Dati provenienti da convertitori A/D

A proposito di ciascuno dei possibili input, dobbiamo porci le seguenti domande:

1. Qual è la sua forma, cioè, che tipo di segnali riceverà realmente il computer?
2. Quando è disponibile l'input e come fa il processore a sapere che è disponibile? Lo dovrà richiedere con un segnale di strobe? L'input dispone di un proprio clock?
3. Per quanto tempo è disponibile?
4. Ogni quanto tempo cambia e in che modo il processore si accorge del cambiamento?
5. L'input consiste di una sequenza o di un blocco di dati? L'ordine è importante?
6. Cosa bisogna fare se i dati contengon degli errori? Può trattarsi di errori di trasmissione, dati errati, sequenze alterate, dati in eccesso, ecc.
7. L'input è correlato ad altri input oppure a degli output?

OUTPUT

L'altro aspetto da definire è quello relativo agli output. Dobbiamo elencare tutti i possibili output prodotti dall'elaboratore. Ecco alcuni esempi:

- Blocchi di dati su linee di trasmissione
- Word di controllo inviate alle periferiche
- Dati inviati a convertitori D/A

A proposito di ciascun output, dobbiamo porci le seguenti domande:

1. Qual è la sua forma, cioè, che tipo di segnali deve inviare il computer?
2. Quando è disponibile l'output e come fa la periferica a sapere che è disponibile?
3. Per quanto tempo è disponibile?
4. Ogni quanto tempo cambia e in che modo la periferica si accorge che è cambiato?
5. Si tratta di una sequenza di più output?
6. Cosa bisogna fare per evitare errori di trasmissione o per rilevare e rimediare ad eventuali guasti di una periferica?
7. L'output è in relazione con degli input o con altri output?

SEZIONE DI ELABORAZIONE

Fra la lettura dei dati in entrata e l'invio dei risultati alle periferiche abbiamo la sezione di elaborazione. A questo punto **è necessario stabilire con esattezza in che modo un computer deve elaborare i dati ricevuti. Le domande sono:**

1. Qual è la procedura (algoritmo) per trasformare i dati d'ingresso in risultati da inviare all'esterno?
2. Esistono delle limitazioni di tempo? Queste possono comprendere la velocità di trasferimento dei dati.
3. Esistono dei limiti riguardo alla memoria utilizzabile per il programma o per i dati, oppure relativamente alle dimensioni dei buffer?
4. Quali programmi standard o tabelle dobbiamo utilizzare? Quali sono le loro caratteristiche?
5. Esistono dei casi speciali? Come devono essere gestiti?
6. Quale deve essere il grado di precisione dei risultati
7. In che modo il programma deve gestire eventuali errori in fase di elaborazione o particolari situazioni, come degli overflow, degli underflow o la perdita di cifre significative?

GESTIONE DEGLI ERRORI

Un aspetto importante in molte applicazioni è la gestione degli errori. Chiaramente, il progettista deve prendere delle precauzioni per premunirsi dagli errori più frequenti e per diagnosticare eventuali guasti. **Ecco alcune delle domande che un progettista deve porsi in fase di definizione del problema:**

1. Quali errori si possono verificare?
2. Quali sono gli errori più probabili? Se è una persona a far funzionare il sistema, l'errore umano è il più frequente. Dopo quello umano, gli errori più comuni sono quelli di comunicazione o di trasmissione, seguiti da quelli meccanici, elettrici, matematici e del processore.
3. Quali errori non saranno riconosciuti immediatamente dal sistema? Un problema particolarmente importante è quello legato al verificarsi di errori che il sistema o l'operatore non riconoscono came tali.
4. In che modo il sistema può ovviare ad eventuali errori nel minor tempo possibile e con la minima perdita di dati, rilevandone sempre la presenza?
5. Quali errori o incovenienti causano comportamenti analoghi da parte del sistema? In che modo si possono distinguere, al fine di ottenere una diagnosi precisa?
6. Quali errori richiedono l'esecuzione di procedure speciali? Ad esempio, degli errori di parità rendono necessaria una nuova trasmissione dei dati?

Un'altra domanda è questa: in che modo un tecnico può individuare sistematicamente, sul posto, l'origine di un funzionamento anomalo, senza dover essere un esperto? Programmi di collaudo incorporati, speciali diagnostiche ed il rilevamento di certi valori (signature analysis) possono essere di aiuto[1].

FATTORI UMANI/INTERAZIONE CON L'OPERATORE

Molti dei sistemi che utilizzano un microprocessore richiedono l'intervento dell'uomo e, in questo caso, **i fattori umani devono essere tenuti presenti durante tutto il processo di sviluppo. Queste sono alcune delle domande che un progettista deve porsi a riguardo:**

1. Quali sono le procedure di input più naturali per un operatore umano?
2. L'operatore è in grado di stabilire facilmente come iniziare, continuare e terminare le operazioni di input?

3. In che modo l'operatore viene informato di eventuali errori procedurali o di un cattivo funzionamento dell'apparecchiatura?
4. Quali sono gli errori più probabili da parte dell'operatore?
5. Come fa l'operatore a sapere che un dato è stato inserito in modo corretto?
6. Le visualizzazioni avvengono in una forma facilmente leggibile e comprensibile da parte dell'operatore?
7. La risposta del sistema è adeguata al tipo di operatore?
8. L'utilizzazione del sistema è abbastanza semplice?
9. Sono disponibili delle indicazioni per l'operatore inesperto?
10. Esistono delle procedure più brevi a disposizione dell'operatore esperto?
11. L'operatore è sempre in grado di determinare lo stato del sistema o di effettuare un reset, dopo eventuali interruzioni o distrazioni?

La realizzazione di un sistema destinato ad un operatore umano è piuttosto complessa. Il processore può rendere il sistema più potente, più flessibile e più rapido. Tuttavia, **è sempre il progettista a dover aggiungere quel tocco di umanità, che accresce notevolmente l'utilità e l'attrattiva del sistema, oltre alla produttività dell'operatore.**

Naturalmente, il processore non ha delle preferenze particolari in situazioni che implicano caratteristiche proprie di un essere umano o, addirittura, delle scelte culturali. Il processore non preferisce uno spostamento da sinistra a destra rispetto ad uno da destra a sinistra, uno spostamento in avanti invece che indietro, un ordine crescente invece di uno decrescente, il sistema decimale anzichè un qualsiasi altro sistema numerico. A differenza di un operatore, non si preoccupa della semplicità, della coerenza, della analogia con esperienze precedenti e di un susseguirsi "logico" delle operazioni. Il processore non si distrae, non si disorienta, non si lascia confondere, nè si annoia. Sono tutti aspetti di cui un progettista deve tener conto, durante la progettazione e lo sviluppo di un sistema interattivo.

ESEMPI

DEFINIRE UN SISTEMA INTERRUTTORE-LUCE

La Figura 16-1 mostra un semplice sistema, in cui l'input proviene da un unico interruttore SPST (Single Pole - Single Throw, unipolare ad una via) e l'uscita è diretta verso un unico LED. In risposta ad una chiusura dell'interruttore, il processore accende il display per un secondo. La definizione di un sistema di questo tipo dovrebbe risultare piuttosto semplice.

Input

Esaminiamo, dapprima, l'input e rispondiamo a ciascuna delle domande elencate in precedenza:

1. L'input è di un solo bit, che può valere '0' (interruttore chiuso) oppure '1' (interruttore aperto).
2. L'input è sempre disponibile e non deve essere richiesto.
3. L'input è ancora disponibile parecchi millisecondi dopo la chiusura.
4. Raramente, l'input cambierà più di una volta ogni pochi secondi. Il processore deve gestire soltanto il rimbalzo dell'interruttore e controllare quando è chiuso.
5. Non c'è un input sequenziale.
6. Errori di input potranno essere un guasto dell'interruttore, un guasto nel circuito d'ingresso o il tentativo dell'operatore di chiudere nuovamente l'interruttore, prima che sia trascorso un intervallo sufficientemente lungo. Spiegheremo in seguito come gestire errori di questo tipo.
7. L'input non dipende da altri input o da eventuali output.

Output

La fase successiva consiste nel prendere in esame l'output. Le risposte alle nostre domande sono:

1. L'output è di un solo bit: '0' per accendere il display, '1' per spegnerlo.
2. Non esistono limitazioni di tempo. La periferica non ha bisogno di essere informata riguardo alla disponibilità o meno del dato.

L'input dell'interruttore è '1' se l'interruttore è aperto, '0' se l'interruttore è chiuso.
La CPU applica l'output al catodo del LED: uno '0' accende il display.

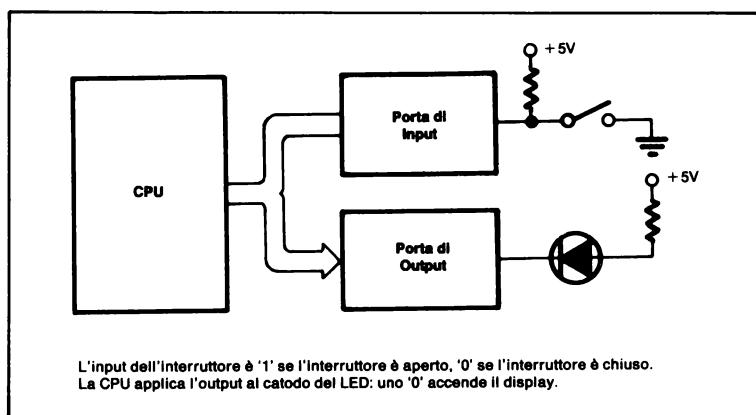



Figura 16-1. Sistema Interruttore - Luce.

3. Se il display è un LED, è sufficiente che il dato sia disponibile solo per pochi millisecondi, con una frequenza dell'impulso pari a 100 volte al secondo, in modo che ad un osservatore il display appaia sempre acceso.
4. Il dato deve cambiare (spegnimento) dopo un secondo
5. Non c'è output sequenziale.
6. Possibili errori di output sono un guasto del display oppure un guasto al circuito di uscita.
7. L'output dipende unicamente dall'input proveniente dall'interruttore e dal tempo di commutazione.

Elaborazione

La fase di elaborazione è estremamente semplice. Non appena l'input proveniente dall'interruttore diventa uno '0' logico, la CPU accende la luce (uno '0' logico) per un secondo. Non esistono limitazioni di tempo o di memoria.

Gestione degli Errori

Esaminiamo adesso gli errori ed i possibili guasti. Possono essere:

- Una nuova chiusura dell'interruttore prima che sia passato un secondo
- Un guasto dell'interruttore
- Un guasto del display
- Un guasto del computer

Il primo è, senz'altro, l'errore più probabile. La soluzione è quella di ignorare le chiusure successive, finchè non è trascorso un secondo. Difficilmente, l'operatore noterà la mancanza di risposta in un periodo così breve. Inoltre, questo elimina la necessità di circuiti e routine per l'eliminazione dei rimbalzi, dal momento che il sistema non reagirà in alcun modo ad eventi del genere.

Chiaramente, gli ultimi tre inconvenienti possono provocare delle conseguenze imprevedibili: il display può restare acceso, spento o cambiare il suo stato casualmente. Questi sono alcuni dei metodi per individuare i guasti:

- Un Lamp-Test per controllare il display; cioè, un pulsante che provochi l'accensione indipendentemente dal processore
- Un collegamento diretto con l'interruttore in modo da controllarne il funzionamento
- Un programma diagnostico che esamini i circuiti di input/ output.

Se il display e l'interruttore funzionano, sarà l'elaboratore ad essere guasto. Un tecnico, fornito di strumenti adeguati, potrà facilmente individuarne la causa.

DEFINIRE UN CARICATORE DI MEMORIA CON INTERRUTTORI

La Figura 16-2 mostra un sistema che consente all'utente di introdurre un dato in una locazione qualsiasi della memoria di un microcomputer. Una porta di input, DPORT, legge il dato da otto interruttori a commutazione. L'altra, CPORT, viene usata per la lettura delle informazioni di controllo. Ci sono quattro interruttori: Indirizzo Alto, Indirizzo Medio, Indirizzo Basso e Dato. L'output è l'ultimo elemento completo proveniente dagli interruttori dei dati; per la visualizzazione sono impiegati otto LED.

Naturalmente, il sistema richiederà anche delle resistenze, dei buffer e dei driver.

Input

Le caratteristiche degli interruttori sono le stesse dell'esempio precedente. Per semplificare la procedura di eliminazione dei rimbalzi e costringere l'operatore a lasciare il pulsante, una volta premuto, dobbiamo fare in modo che il sistema risponda solo dopo che un pulsante è stato rilasciato; è una tecnica comune, che riduce l'usura degli interruttori, poichè l'operatore è meno tentato di premere ripetutamente. In un sistema di questo tipo, **c'è una sequenza di input ben definita:**

Sequenza di inserimento dei dati

1. L'operatore deve posizionare gli interruttori sulla base degli otto bit più significativi di un indirizzo, quindi
2. premere e rilasciare il pulsante Indirizzo Alto. Il valore dei bit comparirà sui diodi luminosi ed il programma interpreterà il dato come il byte alto dell'indirizzo (bit A23-A16).
3. L'operatore deve, quindi, posizionare gli interruttori in base al valore del byte centrale dell'indirizzo (bit A15-A8) e
4. premere e rilasciare il pulsante Indirizzo Medio. I bit compariranno sui diodi luminosi ed il programma considererà il dato come il byte centrale dell'indirizzo.
5. L'operatore deve, poi, posizionare gli interruttori relativamente al valore del byte meno significativo dell'indirizzo (A7-A0) e
6. premere e rilasciare il pulsante Indirizzo Basso. I bit compariranno sui diodi luminosi ed il programma li accetterà come dati relativi al byte basso dell'indirizzo.
7. Infine, l'operatore deve posizionare gli interruttori dei dati in modo che rappresentino il dato da inserire e
8. premere e rilasciare il pulsante Dato. I diodi mostrano, adesso, il valore del dato ed il programma lo salva in memoria all'indirizzo indicato.

Per introdurre un intero programma l'operatore può ripetere varie volte il procedimento. Chiaramente, anche in una situazione

semplificata come questa, dobbiamo considerare molte possibili sequenze. Come proteggersi da sequenze erronee e rendere il sistema facilmente utilizzabile?

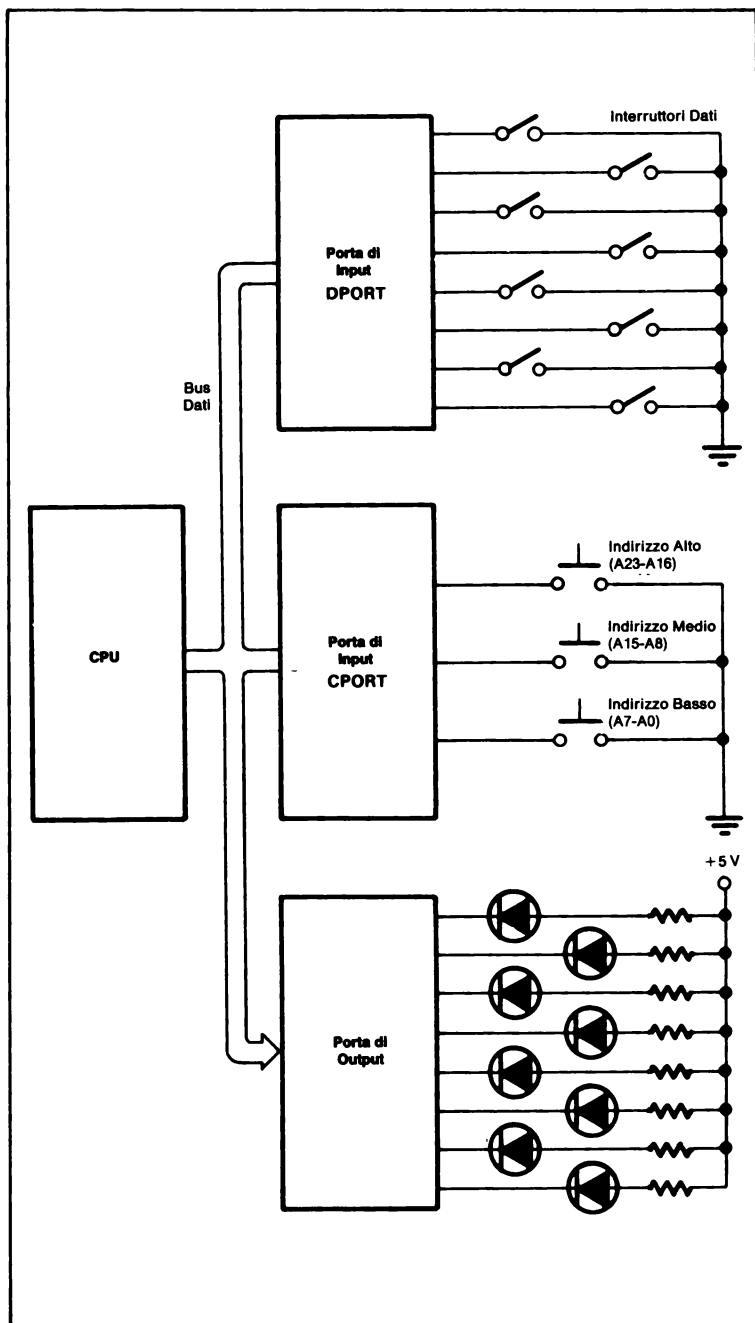



Figura 16-2. Caricatore di Memoria con Interruttori.

Output

L'output non è un problema. Dopo ogni input, il programma invia ai display il complemento dei bit di input (dal momento che i display sono attivi bassi). Il dato in uscita rimane invariato fino alla successiva operazione di input.

Elaborazione

Anche la fase di elaborazione è molto semplice. Non ci sono limitazioni di tempo o di memoria. Il programma elimina i rimbalzi degli interruttori, attendendo alcuni millisecondi, e fornisce ai display dei dati complementati.

Gestione degli Errori

Gli errori più frequenti sono dovuti a sbagli dell'operatore. Può trattarsi di:

- Inserimenti errati
- Una sequenza non corretta
- Inserimenti incompleti; ad esempio, dimenticare di inserire il dato.

Il sistema deve essere in grado di gestire questi problemi in modo opportuno, poichè, in fase operativa, si verificheranno quasi sicuramente.

Il progettista deve tener conto anche delle conseguenze di un guasto alle apparecchiature. Come nel caso precedente, le possibili difficoltà sono:

- Un guasto ad un interruttore
- Un guasto ad un display
- Un guasto al computer

In un sistema come questo, tuttavia, dobbiamo prestare attenzione, soprattutto, al modo in cui questi inconvenienti agiscono sul sistema. Un guasto del computer provocherà il blocco del'intero sistema, che sarà facilmente rilevabile. Il guasto di un display può non essere rilevato immediatamente, per cui è opportuna la presenza di un Lamp-Test per un'eventuale verifica. Per diagnosticare il caso in cui le linee di uscita fossero cortocircuitate fra loro, sarebbe preferibile poter controllare separatamente i singoli LED. Inoltre, anche il guasto di un interruttore può non essere rilevato immediatamente dall'operartore, che, una volta resosene conto, dovrebbe localizzarlo mediante un processo di eliminazione.

Correzione degli Errori dell'Operatore

Esaminiamo **alcuni possibili errori dell'operatore.** I più comuni saranno:

- Dati errati
- Una sequenza errata di inserimenti o di interruttori
- Passare all'inserimento successivo, senza aver completato quello precedente

Procedure di correzione degli errori

È presumibile che l'operatore noti dei dati errati non appena appaiono sui display. Quale potrebbe essere una adeguata procedura di correzione? Queste sono alcune fra quelle possibili:

1. Qualora si verificasse un errore in relazione all'Indirizzo Alto, l'operatore dovrà completare ugualmente la procedura d'inserimento; cioè, inserire Indirizzo Medio, Indirizzo Basso e Dato. Chiaramente, questa è una procedura lunga e fastidiosa.

2. L'operatore può ricominciare tutta la sequenza, ritornando alla fase di inserimento dell'Indirizzo Alto. Questa soluzione si rivela particolarmente efficace, se l'errore ha avuto luogo in corrispondenza dell'Indirizzo Alto, ma, qualora, si sia verificato con l'Indirizzo Medio o Basso o al momento di inserire il Dato, l'operatore sarà costretto ad introdurre di nuovo anche tutti i dati precedenti.

3. L'operatore può introdurre i dati in un ordine qualsiasi. È sufficiente che prima abbia premuto il pulsante corrispondente.In questo modo diventa possibile fare delle correzioni in ogni momento.

Questo tipo di procedura è preferibile ad una che non consenta una correzione immediata dell'errore, che abbia varie possibilità di uscita oppure inserisca i dati nel sistema senza consentire all'operatore un ultimo controllo. In certi casi, per aumentare l'efficienza dell'operatore è consigliabile l'aggiunta di altre componenti hardware o il miglioramento del software. È sempre meglio lasciare il lavoro più noioso ed il riconoscimento di sequenze arbitrarie all'elaboratore, che non si stanca mai e non dimentica le procedure esatte.

Un ulteriore aiuto potrebbero essere alcune luci di stato, che indichino il significato di ciò che viene visualizzato. Quattro luci di stato, contrassegnate "Indirizzo Alto", "indirizzo Medio", "Indirizzo Basso" e "Dato", segnaleranno all'operatore ciò che è stato inserito, senza che debba ricordarsi quale pulsante ha premuto. È il processore a dover controllare la sequenza; la maggiore complessità del programma che questo comporta semplificherà notevolmente il compito dell'operatore. Chiaramente, quattro gruppi separati di

display, oltre alla possibilità di esaminare una locazione di memoria, risulterebbero ancora più utili.

Benchè si sia voluto porre l'accento sull'interazione con un essere umano, quella con una macchina o un sistema ha delle caratteristiche molto simili. È l'elaboratore che dovrebbe fare gran parte del lavoro. Aumentando la complessità delle funzioni svolte dal microprocessore diventa semplice ripristinare il sistema dopo un errore e le cause di un guasto risulteranno molto più evidenti; inoltre l'intero sistema funzionerà meglio e la manutenzione sarà più facile. Non bisogna aspettare di aver realizzato il software per prendere in considerazione gli aspetti relativi all'uso ed alla manutenzione di un sistema: ne va tenuto conto fin dalla fase di definizione del problema.

DEFINIRE UN TERMINALE DI VERIFICA

La Figura 16-3 è un diagramma a blocchi di un semplice terminale per la verifica delle carte di credito. Una prima porta di input riceve i dati da una tastiera (cfr. Figura 16-4); un'altra porta riceve i dati necessari alla verifica da una linea di trasmissione. Una porta di output invia i dati ad un gruppo di display (cfr. Figura 6-5), mentre un'altra invia il numero della carta di credito al computer centrale. Una terza porta di output accende una luce ogni volta che il terminale è pronto per accettare una richiesta ed un'altra luce quando l'operatore invia le informazioni. La luce che indica "Occupato" si spegne nel momento in cui il terminale riceve una risposta. Sia l'input che l'out-

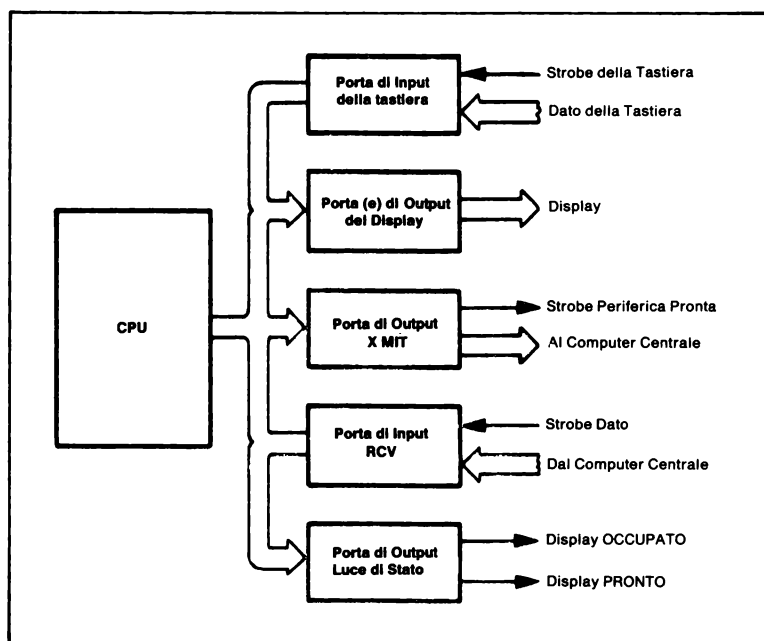
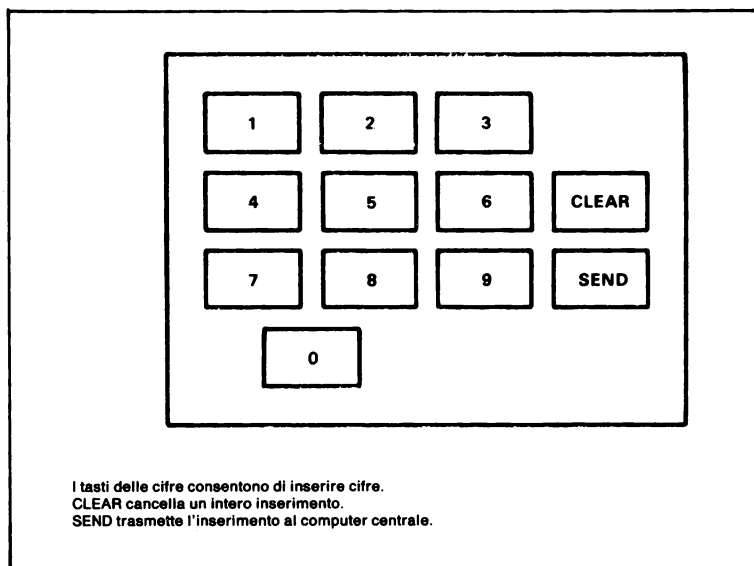



Figura 16-3. Diagramma a Blocchi per un Terminale di Verifica.

Figura 16-4. Tastiera di un Terminale di Verifica.

put sono, chiaramente, più complessi che nel caso precedente, benchè il tipo di elaborazione sia ancora molto semplice.

Possono essere impiegati dei display addizionali per evidenziare meglio il significato della risposta. Molti terminali usano una luce verde in caso di risposta positiva, una luce rossa nel caso di una risposta negativa ed una luce gialla per indicare di rivolgersi al responsabile del negozio. Queste luci dovranno sempre essere contrassegnate con il loro significato nell'eventualità di operatori daltonici.

Input

Esaminiamo prima di tutto l'input dalla tastiera. Naturalmente, è diverso dall'input proveniente dall'interruttore, dal momento che la

Figura 16-5. Display di un Terminale di Verifica.

CPU deve avere la possibilità di distinguere un nuovo dato. **Si parte dal presupposto che ogni chiusura di tasto fornisca un codice esadecimale diverso (si può codificare ciascuno dei 12 tasti con una sola cifra) ed uno strobe. Il programma dovrà riconoscere lo strobe e prelevare il numero esadecimale corrispondente al tasto premuto.** Esiste una limitazione di tempo, in quanto il programma non può perdere nessun dato e nessuno strobe. Non si tratta di una grossa limitazione, poichè come minimo i tasti saranno premuti alla distanza di parecchi millisecondi l'uno dall'altro.

Formato dell'input e procedura di trasmissione

Anche l'input di trasmissione consiste di una serie di caratteri, ciascuno identificato da uno strobe (magari proveniente da un UART). Il programma dovrà riconoscere ogni strobe e prelevare il carattere. I dati, essendo inviati attraverso una linea di trasmissione, saranno organizzati sotto forma di messaggi, il cui formato potrebbe essere questo:

- Caratteri introduttivi (o intestazione)
- Indirizzo del terminale di destinazione
- Un sì o un no codificati
- Caratteri terminali o di coda

Il terminale deve verificare l'intestazione, leggere l'indirizzo di destinazione e controllare se il messaggio è destinato a lui. In caso affermativo, accetta il dato. L'indirizzo potrebbe essere ottenuto (e spesso lo è) tramite connessioni hardware, in modo che il terminale riceva solo i messaggi che gli sono destinati. Questa soluzione semplifica il software necessario, privando, però, il sistema di una certa flessibilità.

Output

Anche l'output è molto più complesso, rispetto agli esempi precedenti. **Se i display sono multiplexati, il processore non può limitarsi ad inviare un dato alla relativa porta di output, ma deve anche selezionare un display in particolare.** Avremo bisogno, a questo scopo, di una porta di controllo separata oppure di un contatore e di un decoder. Dei controlli hardware provvedono ad eliminare gli zeri iniziali, finchè la prima cifra non è un numero diverso da zero. Questo può essere ottenuto anche tramite software. Le limitazioni di tempo riguardano la lunghezza dell'impulso e la frequenza necessaria a produrre una visualizzazione che appaia stabile all'operatore.

Formato dell'output

L'output sulla linea di comunicazione consisterà di una serie di caratteri con un formato particolare. Il programma dovrà anche tener conto della necessità di un intervallo fra un carattere e l'altro. Un possibile formato per il messaggio di output è il seguente:

- Intestazione
- Indirizzo del terminale
- Numero della carta di credito
- Coda

Un computer centrale, per lo smistamento delle comunicazioni può controllare i vari terminali, per verificare se ci sono dei dati pronti per essere inviati.

Elaborazione

In un sistema di questo tipo, l'elaborazione comporta nuove funzioni:

- Identificare i tasti di controllo in base al numero ed eseguire le relative operazioni
- Aggiungere l'intestazione, l'indirizzo del terminale ed i dati di coda al messaggio in uscita
- Riconoscere l'intestazione e la coda del messaggio di ritorno
- Controllare l'indirizzo del terminale

Nessuna di queste funzioni comporta dei calcoli complessi o delle particolari limitazioni di tempo e di memoria.

Gestione degli Errori

In questo sistema anche il numero dei possibili errori è, naturalmente, molto maggiore rispetto agli esempi precedenti. Consideriamo, per prima cosa, gli errori che può compiere un operatore:

- Inserire in modo errato il numero della carta di credito
- Inviare un numero incompleto
- Inviare un altro numero, mentre il computer centrale sta verificandone uno precedente
- Azzerare degli elementi non esistenti

Alcuni di questi errori si possono gestire facilmente con una corretta organizzazione. Ad esempio, il programma non accetterà il tasto Send (Invio) finchè il numero della carta di credito non è stato inserito interamente, e dovrà ignorare ogni successivo inserimento dalla tastiera se non ha ancora ricevuto una risposta dal computer centrale. L'operatore saprà che il nuovo inserimento non è stato accettato dal fatto che la luce "Occupato" non si sarà accesa. Si accorgerà anche quando la tastiera è stata esclusa (il programma ignora eventuali inserimenti dalla tastiera), dal fatto che i dati introdotti non compariranno sul display e la luce "Pronto" sarà spenta.

Correggere gli Errori di Tastiera

Un problema è evidentemente anche quello di eventuali inserimenti errati. **Se l'operatore si rende conto di aver fatto un errore può servirsi del tasto Clear per fare delle correzioni. L'operatore, probabil-**

mente, troverà più comodo poter disporre di due tasti Clear, uno che cancella l'ultimo tasto premuto e l'altro che cancella l'intero inserimento. Questo si rivela utile sia quando l'operatore si accorge immediatamente dell'errore, sia quando se ne rende conto solo in una fase successiva della procedura. **Egli dovrebbe essere capace di rilevare l'errore immediatamente, in modo da non dover ricominciare tutto dall'inizio. È probabile, comunque, che un operatore commetta degli errori senza rendersene conto. I numeri delle carte di credito contengono, in genere, una cifra di controllo, per cui il terminale potrebbe verificarne l'esattezza prima che i dati siano inviati al computer centrale,** che eviterebbe di perdere del tempo nel controllo.

È necessario, tuttavia, che il terminale disponga di un mezzo per informare l'operatore dell'errore, magari accendendo e spegnendo uno dei display in modo intermittente o fornendo qualche altro segnale che venga sicuramente notato dall'operatore.

Un altro problema è quello di informare l'operatore che un inserimento è andato perso o è stato elaborato in modo sbagliato. Alcuni terminali, dopo un certo intervallo, si sbloccano da soli. L'operatore osserva che la luce di "Occupato" si è spenta senza aver ottenuto una risposta, e, quindi, si presuppone che ripeta l'inserimento. Dopo un paio di tentativi andati a vuoto, informerà del problema il personale incaricato.

Sono frequenti anche i guasti alle apparecchiature. Oltre ai display, alla tastiera ed al processore, in questo caso c'è anche la possibilità di inconvenienti o guasti agli apparecchi di comunicazione o al computer centrale.

Correggere gli Errori di Trasmissione

La trasmissione dei dati dovrà, probabilmente, includere un controllo per evitare errori e alcune procedure di correzione. Questi sono alcuni dei metodi possibili:

1. La parità serve a individuare un errore, ma non a correggerlo. Il ricevitore dovrà, in qualche modo, chiedere la ripetizione dell'invio, mentre il trasmettitore dovrà conservare una copia del dato, finchè non gli è stato comunicato che la ricezione è avvenuta correttamente. Si tratta, comunque, di un procedimento facilmente realizzabile.
2. Messaggi brevi utilizzano schemi più complessi. Ad esempio, la risposta si/no al terminale potrebbe essere codificata in modo tale che sia possibile, oltre ad individuare un errore, anche correggerlo.
3. Un codice di riconoscimento oppure un certo numero di nuovi tentativi andati a vuoto potrebberero far scattare un segnale, che informi l'operatore di un guasto alle apparecchiature di comunicazione (impossibilità di trasferire il messaggio senza errori) o all'elaboratore centrale (nessuna risposta entro un

certo periodo di tempo). Uno schema simile, insieme con un Lamp Test, consentirebbe una diagnosi rapida degli inconvenienti.

Un segnale che indichi un guasto agli apparecchi di comunicazione o al computer centrale dovrebbe anche sbloccare il terminale, permettergli, cioè, di accettare un nuovo inserimento.. Questo è indispensabile, se non è possibile introdurre nuovi dati mentre è in corso una verifica. **Il terminale si può anche sbloccare da solo dopo un determinato periodo di tempo. Certi inserimenti potrebbero essere riservati a scopi diagnostici;** cioè, alcuni numeri di carte di credito potrebbero essere usati per controllare il funzionamento interno del terminale o per effettuare un test dei display.

CONCLUSIONI

La definizione del problema è un aspetto importante nello sviluppo del software, come lo è in tutte le realizzazioni tecniche. Non c'è bisogno di programmare e neanche di conoscere l'elaboratore utilizzato; è sufficiente, invece, avere un'idea chiara del sistema e delle buone capacità di valutazione. I microprocessori mettono a disposizione flessibilità ed intelligenza specifica, che il progettista, poi, deve saper utilizzare per ottenere una vasta gamma di prestazioni.

La definizione del problema è indipendente dal particolare tipo di elaboratore, dal linguaggio e dal sistema di sviluppo utilizzati. Dovrebbe, tuttavia, fornire delle indicazioni [illegible]po e la velocità del computer richiesto da una determinata applicazione e sul rapporto ottimale di risorse hardware e software necessarie. La fase di definizione del problema deve, in teoria, prescindere anche dal fatto che venga utilizzato un computer o meno, anche se una conoscenza delle possibilità che questo è in grado di offrire, serve al progettista per stabilire le modalità di realizzazione delle varie procedure.

BIBLIOGRAFIA

1. D.R. Ballard. "Designing Fail-Safe Microprocessor Systems," *Electronics* January 4, 1979, pp. 139-43.
"A Designer's Guide to Signature Analysis," Hewlett-Packard Nota Applicativa 222, Hewlett-Packard, Inc, Palo Alto, CA, 1977.
Donn, E.S. and M.D. Lippman "Efficient and Effective Microcomputer Testing Requires Careful Preplanning," *EDN*, February 20, 1979, pp. 97-107 (comprende esempi di autotest per il 6502).

Gordon, G. and H. Nadig. "Hexadecimal Signatures Identify Troublespots in Microprocessor Systems," *Electronics*, March 3, 1977, pp. 89-96.
Neil, M. and R. Goodner. "Designing a Serviceman's Needs into Microprocessor Based Systems," *Electronics*, March 1, 1979, pp. 122-28.
Schweber, W. and L. Pearce. "Software Signature Analysus Identifies and Checks PROMs," *EDN*, November 5, 1978, pp. 79-81.
Srini, V.P. *Fault Diagnosis of Microprocessor Systems," *Computer*, January 1977, pp. 60-65.

2. Per una breve analisi dei fattori umani cfr. G. Morris. "Make Your Next Instruments Design Emphasize User Needs and Wants," *EDN*, October 20, 1978, pp. 100-05.

CAPITOLO 17

PROGETTAZIONE DI UN PROGRAMMA

È la fase in cui un problema, già definito, viene espresso sotto forma di programma. Nel caso di programmi brevi e piuttosto semplici, è sufficiente scrivere un diagramma di flusso di una pagina, non di più. Se il programma, invece, è più lungo e complesso, è necessario far ricorso a dei metodi più elaborati.

Tecniche di progettazione

In questo capitolo prenderemo in esame i diagrammi di flusso, la programmazione modulare, la programmazione strutturata e la progettazione top-down. Cercheremo di analizzare i motivi che rendono necessaria l'adozione di queste tecniche, i loro vantaggi e svantaggi. Non abbiamo intenzione, tuttavia, di privilegiare un metodo rispetto ad un altro, dal momento che non ne esiste uno migliore in assoluto. Il nostro obiettivo è quello di realizzare un sistema valido e funzionante in modo corretto, non di seguire scrupolosamente le regole di questa o quella metodologia.

PRINCIPI FONDAMENTALI

Tutte le varie procedure si basano su dei principi comuni, validi per qualsiasi tipo di progetto. Eccone alcuni:

1. **Procedere a piccoli passi.** Non cercare di fare troppe cose tutte in una volta.
2. **Suddividere dei lavori complessi in compiti più semplici, logicamente distinti.** Rendere queste funzioni indipendenti l'una dall'altra, in modo da poterle collaudare separatamente e modificarne una senza compromettere le altre.
3. **Il flusso di controllo deve essere semplice,** per seguire meglio le varie sequenze presenti in un programma e poter localizzare gli errori.
4. **Per definire la struttura di un programma utilizzare disegni e grafici,** più facili da visualizzare rispetto alle parole. Questo è il grande vantaggio dei diagrammi di flusso.
5. **Cercare di ottenere il massimo di chiarezza e di semplicità.** Se necessario, una volta che il sistema funziona se ne potranno migliorare le prestazioni.

6. **Procedere in modo sistematico,** servendosi di liste di controllo e di procedure standardizzate.
7. Non sfidare il destino. **Non usare metodi di cui non si è veramenti sicuri o servirsene con molta discrezione. Fare attenzione alle situazioni che possono creare confusione** e chiarirle non appena possibile.
8. **Il sistema ha bisogno di una verifica, di un collaudo e di una manutenzione, tutti aspetti di cui occorre tener conto fin dall'inizio.**
9. **Usare sempre termini e metologie molto semplici.** Durante la progettazione di un programma, le ripetizioni non sono un errore e la complessità non è una virtù.
10. **Formulare interamente il progetto prima di iniziare la fase di codifica.** Non bisogna cedere alla tentazione di cominciare a scrivere le istruzioni; sarebbe come fare gli elenchi dei componenti di un circuito e preparare la scheda, prima di sapere esattamente ciò che è necessario.
11. Occorre particolare attenzione nelle parti soggette ad eventuali modifiche, **la cui realizzazione dovrà risultare estremamente semplice.**
12. **Tenere sempre presente il quadro complessivo.** Realizzare, inizialmente, una struttura generale, di cui potranno essere definite e collaudate le singole parti. Non rimandare l'integrazione dell'intero sistema alla fase conclusiva.
13. Se i dati sono complessi e strettamente correlati tra loro **occorre organizzarli con la stessa cura del programma. Alla fine di questo capitolo, analizzeremo brevemente la progettazione delle strutture dati.**

Diagrammi di dati

DIAGRAMMI DI FLUSSO

L'impiego dei diagrammi di flusso è certamente il più conosciuto fra tutti i metodi di progettazione. I manuali di programmazione consigliano di scrivere, prima di tutto, dei diagrammi di flusso completi e, successivamente, il programma vero e proprio. In realtà, sono pochi i programmatori che lavorano in questo modo e i diagrammi di flusso sono considerati spesso un inutile fastidio. Cercheremo di descriverne vantaggi e svantaggi e di mostrare il posto che questa tecnica occupa nella progettazione di un programma.

VANTAGGI DEI DIAGRAMMI DI FLUSSO

Il vantaggio fondamentale di un diagramma di flusso è il fatto che si tratta di una rappresentazione grafica. È possibile visualizzare l'intero sistema e rendersi conto, allo stesso tempo, delle correlazioni

esistenti fra le varie componenti: gli errori di logica e le incoerenze saltano subito all'occhio. Un diagramma di flusso, se fatto bene, fornisce un quadro complessivo del sistema.

Altri particolari vantaggi dei diagrammi di flusso sono:

1. Esistono dei simboli standard (cfr. Figura 17-1) che tutti adottano per convenzione.
2. I diagrammi di flusso sono comprensibili anche a chi non ha una grande esperienza di programmazione.
3. I diagrammi di flusso possono essere usati per suddividere l'intero progetto in sottoinsiemi più semplici e, quindi, poter avere un'idea dei progressi compiuti in fase di realizzazione.
4. I diagrammi di flusso mostrano la sequenza delle operazioni ed aiutano, quindi, ad individuare le possibili cause di un errore.
5. I diagrammi di flusso sono largamente impiegati anche in settori diversi dalla programmazione.
6. Esistono molti strumenti che servono alla realizzazione dei diagrammi di flusso, fra cui package grafici con simboli predefiniti.

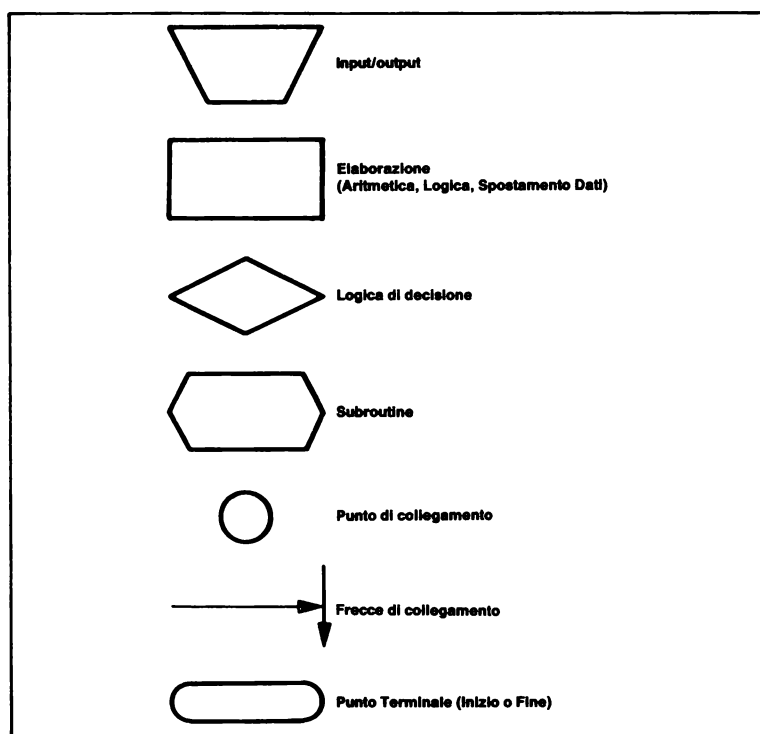



Figura 17-1. Simboli Standard dei Diagrammi di Flusso.

SVANTAGGI DEI DIAGRAMMI DI FLUSSO

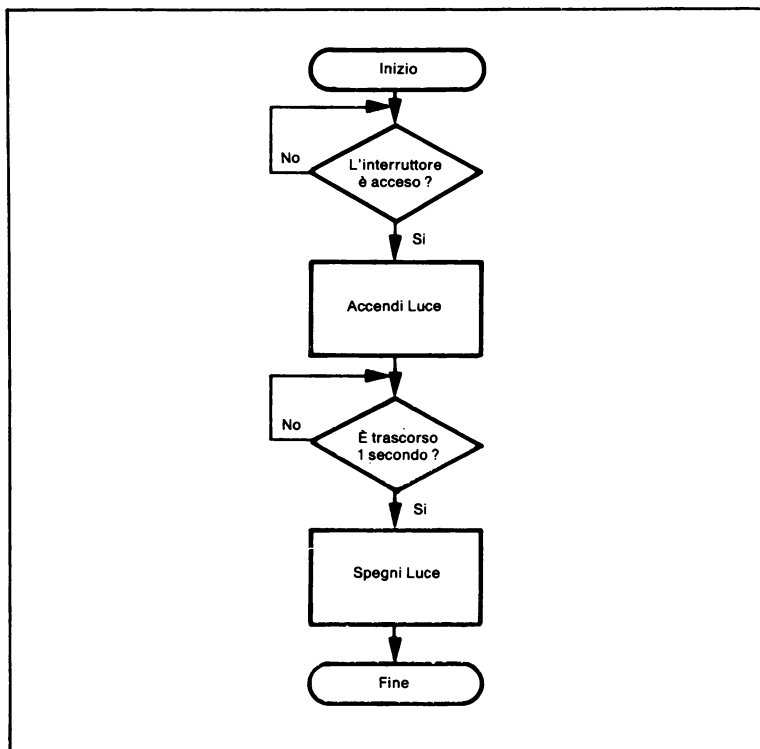
Quelli che abbiamo appena visto sono tutti vantaggi molto importanti ed i diagrammi di flusso continueranno, senza dubbio, ad essere largamente utilizzati. Ma **questa tecnica presenta anche alcuni svantaggi nell'ambito della programmazione:**

1. I diagrammi di flusso sono difficili da progettare, disegnare e modificare, tranne quando sono molto semplici.
2. Non c'è la possibilità di effettuare il debug o il collaudo di un diagramma di flusso.
3. I diagrammi di flusso finiscono spesso per essere poco chiari. È difficile stabilire la quantità di dettagli necessari perchè il diagramma sia veramente utile e non diventi, invece, quasi un listato di programma.
4. I diagrammi di flusso mostrano solo l'organizzazione del programma, senza fornire indicazioni relative alla struttura dei dati e dei moduli di input/output
5. I diagrammi di flusso non sono di aiuto con l'hardware o con problemi di temporizzazione, nè danno indicazioni su dove questi possano eventualmente verificarsi.
6. I diagrammi di flusso consentono una progettazione non strutturata. Non ci sono regole che indichino il numero di input/output, il numero ed il tipo di interconnessioni o la logica che può essere impiegata.
7. Non esiste un modo sufficientemente chiaro di rappresentare la semplice ripetizione di un loop.

COME RENDERE UTILI I DIAGRAMMI DI FLUSSO

I diagrammi di flusso si rivelano più utili se ignorano le variabili di un programma e formulano domande dirette. Naturalmente, spesso sono necessari dei compromessi. **Qualche volta è opportuno addirittura disporre di due versioni dello stesso diagramma: una versione generica in un linguaggio non da iniziati, utile a chi non è un programmatore, ed una versione destinata al programmatore che utilizza le variabili del programma.**

In certi casi anche la stesura di un terzo tipo di diagramma di flusso, riservato ai dati, può essere di aiuto. Esso costituisce anche un riferimento per gli altri diagrammi, in quanto mostra come il programma gestisce un determinato tipo di dati. I normali diagrammi di flusso indicano in che modo il programma procede attraverso le varie fasi operative, provvedendo a gestire dati diversi. **I diagrammi dei dati,** invece, **chiariscono in che modo i dati sono utilizzati all'interno del sistema, passando da una parte all'altra del programma;** si rivelano molto utili in fase di debugging e di manutenzione, dal

Figura 17-2. Diagramma di Flusso di una Risposta di un Secondo ed un Interruttore.

momento che spesso gli errori si manifestano con la errata gestione di un particolare tipo di dati.

Perciò, **la tecnica dei diagrammi di flusso può rivelarsi utile, ma è consigliabile non eccedere. Serve, senza dubbio, a documentare meglio un programma, in quanto utilizza dei simboli standard, comprensibili anche a coloro che non sono programmatori.** Come strumento di progettazione, tuttavia, i diagrammi di flusso forniscono solamente un'indicazione generale; **un programmatore, inoltre, non ha la possibilità di fare il debug di un diagramma molto dettagliato,** che spesso è più difficile da progettare del programma stesso.

ESEMPI

Diagramma di Flusso di un Sistema Interruttore - Luce

È facile realizzare il diagramma di flusso di questa funzione piuttosto semplice, in cui un singolo interruttore accende una luce per un secondo. Funzioni di questo tipo servono come esempi in tutti i manuali che affrontano l'argomento dei diagrammi di flusso, ma raramente si presentano nella pratica situazioni come questa. La struttura dei dati è così semplice che può essere tranquillamente ignorata.

La Figura 17-2 mostra il diagramma di flusso. Non è difficile stabilire quanto debba essere dettagliato. Esso ci dà un quadro immediato della procedura, comprensibile a chiunque.

Diagramma di Flusso di un Caricatore di Memoria con Interruttori

Questo sistema (cfr. Figura 16-2) è notevolmente più complesso di quello dell'esempio precedente e comporta molte decisioni in più. **Il diagramma di flusso è più difficile da disegnare e non è altrettanto comprensibile.** In questo caso, ci troviamo di fronte anche al problema dell'impossibilità di verificare e collaudare un diagramma.

Il diagramma di flusso della Figura 17-3 comprende i miglioramenti che abbiamo suggerito durante la definizione del problema. Chiaramente, questo diagramma comincia a diventare troppo detta-

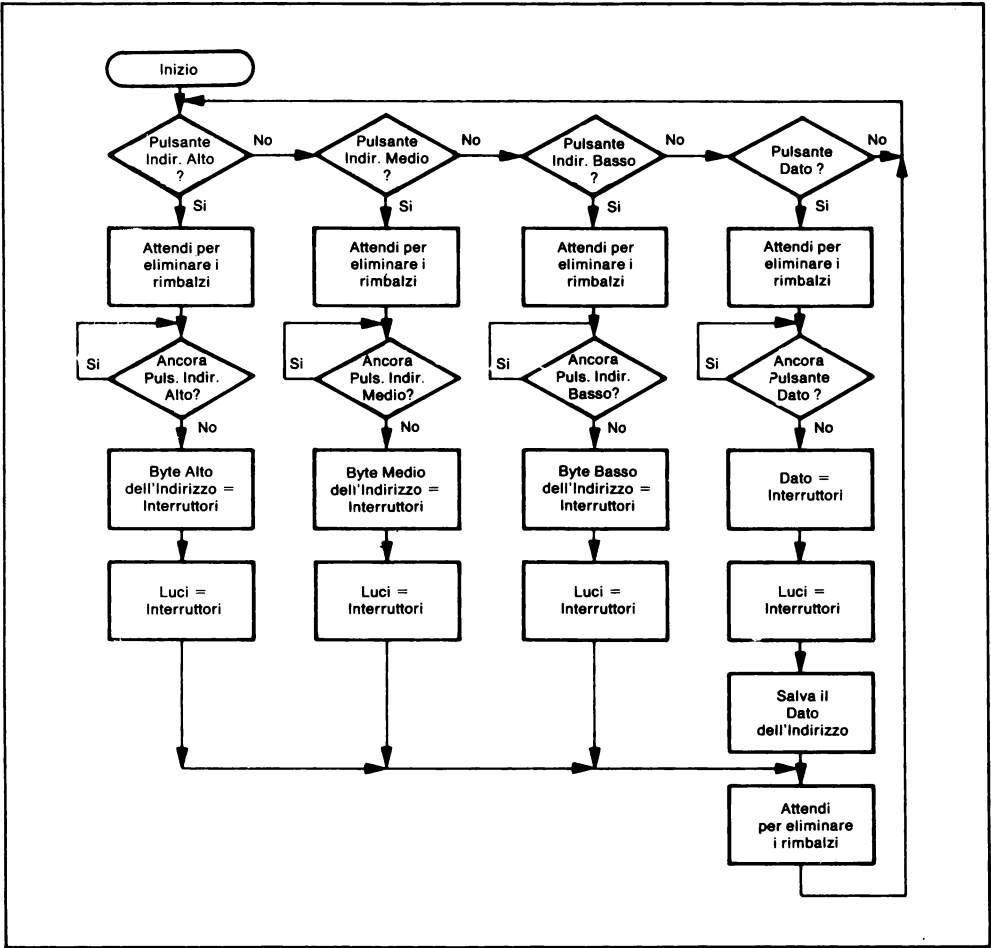



Figura 17-3. Diagramma di Flusso di un Caricatore di Memoria con Interruttori

gliato e non è molto più utile di una descrizione scritta. L'aggiunta di altre caratteristiche (luci di stato che indicano il significato degli inserimenti e consentono all'operatore un controllo, una volta immessi i dati) lo renderebbero ancora più complesso. Disegnare un diagramma completo, partendo da zero, finisce per essere un compito veramente arduo. Comunque, una volta che il programma è stato scritto , il diagramma di flusso è utile come documentazione.

Diagramma di Flusso di un Terminale di Verifica

In questa applicazione (cfr. Figure16-3 /16-5) il diagramma di flusso è ancora più complesso che nel caso del caricatore di memoria. **La soluzione migliore è quella di fare dei diagrammi separati per le varie sezioni,** in modo che essi non diventino mai troppo complessi. Tuttavia, la presenza di strutture dati (come nel caso del display e dei messaggi) aumenterà eccessivamente la differenza fra il diagramma ed il programma.

Esaminiamo alcune di queste sezioni. **La Figura 17-4 mostra il processo di inserimento da tastiera relativamente ai tasti delle cifre.** Il

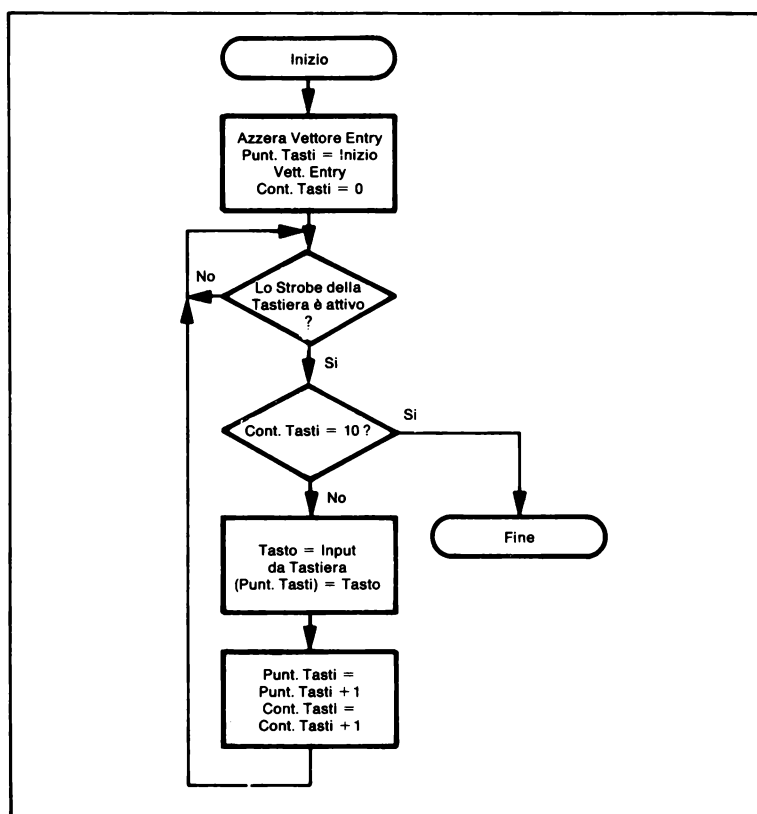



Figura 17-4. Diagramma di Flusso del Processo di Inserimento da Tastiera.

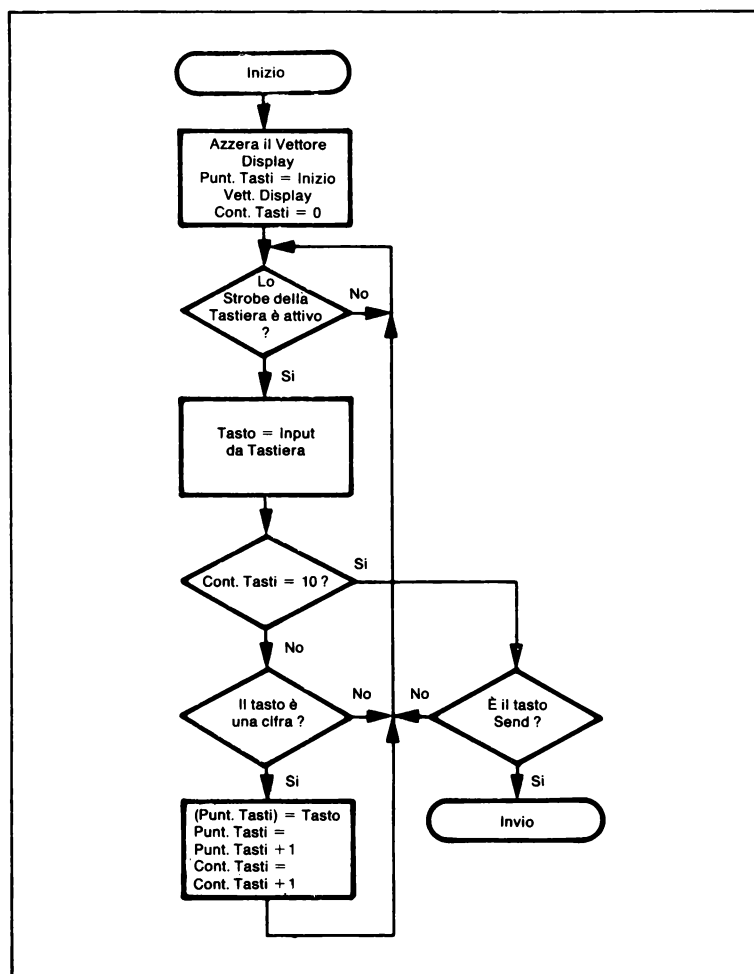



Figura 17-5. Diagramma di Flusso del Processo di Inserimento con il Tasto Send.

programma deve prelevare il dato dopo ogni strobe e mettere la cifra nel vettore del display, se c'e ancora spazio. Se nel vettore vi sono già dieci cifre, il programma deve semplicemente ignorare l'inserimento.

Il programma deve gestire contemporaneamente tutti i display. Inoltre, il software o l'hardware disattiveranno lo strobe dalla tastiera dopo che il processore ha letto una cifra.

La Figura 17-5 aggiunge il tasto Send (Invio), che, naturalmente, è del tutto opzionale. Il terminale potrebbe benissimo inviare un dato non appena l'operatore ha introdotto un numero completo. Tuttavia, una procedura di questo genere non darebbe la possibilità di controllare il numero, dopo averlo inserito. Il diagramma con il tasto Send è più complesso, perchè ci sono due alternative:

1. Se l'operatore non ha inserito dieci cifre, il programma ignora il tasto Send e considera un qualunque altro tasto come parte dell'inserimento.
2. Se l'operatore ha inserito dieci cifre, il programma risponde al tasto Send trasferendo il controllo alla relativa routine ed ignorando tutti gli altri tasti.

A questo punto, il diagramma è diventato già molto più difficile da organizzare e da seguire e non c'è neanche il modo di controllarne l'esattezza.

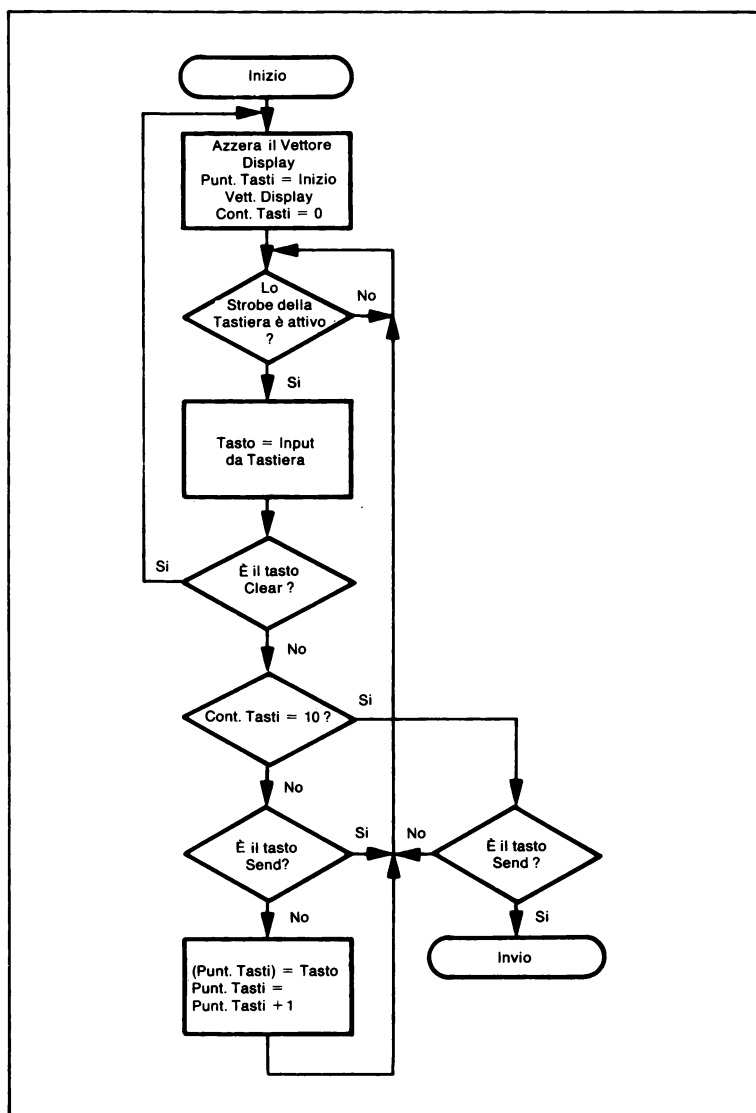



Figura 17-6. Diagramma di Flusso del Processo di Inserimento con Tasti Funzione.

La Figura 17-6 mostra il diagramma di flusso della procedura di inserimento da tastiera, con tutti i tasti funzione. In questo esempio, il flusso di controllo non è semplice ed è necessario inserire qualche descrizione scritta. L'organizzazione e la stesura di diagrammi complessi richiede una attenta pianificazione. Un buon metodo è quello di aggiungere nuove caratteristiche una per volta, come abbiamo fatto adesso, ma questo comporta un notevole numero di disegni in più. Inoltre, non dobbiamo dimenticare che, durante tutto il processo di inserimento dalla tastiera, il programma deve anche attivare i display, nel caso che questi siano multiplexed e non sotto il controllo di registri di shift o di altro hardware.

La Figura 17-7 è il diagramma di flusso di una routine di ricezione. Si presuppone che la conversione seriale-parallelo ed il controllo degli errori avvengano tramite hardware (ad es., un UART). Il processore deve:

1. Cercare l'intestazione. (Si presuppone che si tratti di un solo carattere).
2. Leggere l'indirizzo di destinazione (deve avere una lunghezza di tre caratteri) e vedere se il messaggio è destinato a questo terminale; cioè, se i tre caratteri concordano con l'indirizzo del terminale.
3. Attendere il carattere di coda
4. Se il messaggio è destinato al terminale, spegnere la luce di "Occupato" e passare alla routine di Risposta sul Display.
5. In caso di errore, chiedere la ripetizione della trasmissione, andando all'apposita routine RTRANS.

Questa routine comporta un gran numero di decisioni ed il diagramma di flusso non è nè semplice, nè facilmente comprensibile.

Chiaramente, siamo andati parecchio avanti dal primo semplice diagramma (cfr. Figura 17-7). Un gruppo completo di diagrammi di flusso per un terminale di transazione sarebbe un compito veramente arduo. Consisterebbe di parecchi schemi correlati fra loro con una logica molto complessa e richiederebbe uno sforzo enorme, quasi come scrivere un programma preliminare. Non sarebbe, però, altrettanto utile, dal momento che non sarebbe possibile verificare dei diagrammi sul computer.

PROGRAMMAZIONE MODULARE

Stesura di diagrammi complessi

Quando i programmi sono lunghi e complessi i diagrammi di flusso non sono più uno strumento di progettazione soddisfacente. Tuttavia, la definizione del problema e i diagrammi di flusso rendono possibile suddividere il programma in varie parti di minore complessità. **La suddivisione del programma in sottosezioni o moduli è**

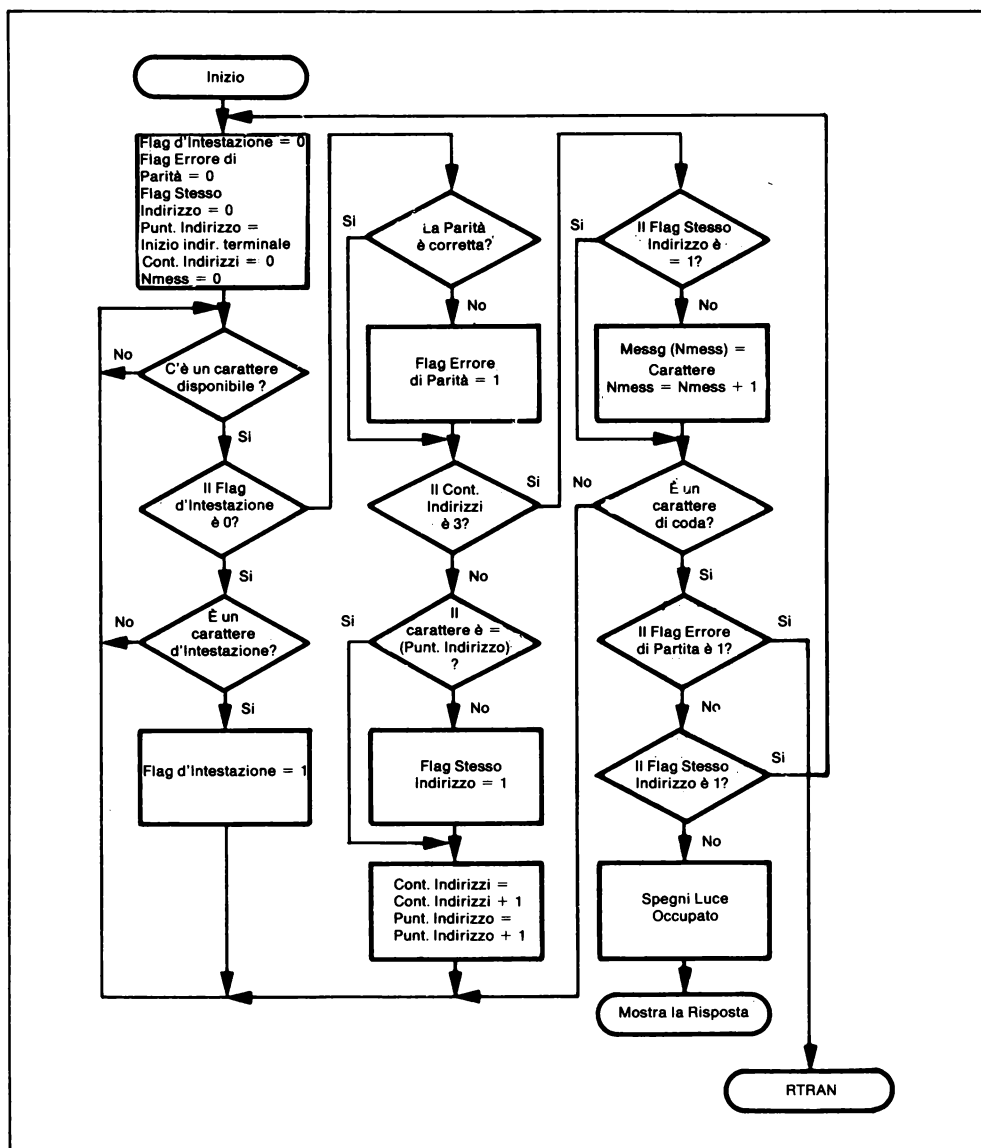



Figura 17-7. Diagramma di Flusso del Processo di una Routine di Ricezione

Definizione di "Programmazione Modulare"

chiamata "programmazione modulare". Come è facilmente intuibile, gli esempi che abbiamo presentato nei capitoli precedenti sono per lo più moduli di un programma più ampio. **I problemi che un progettista deve affrontare, nel caso della programmazione modulare, sono la suddivisione del programma in moduli e la loro successiva integrazione in un insieme unico.**

VANTAGGI DELLA PROGRAMMAZIONE MODULARE

I vantaggi della programmazione modulare sono abbastanza evidenti:

1. È più facile scrivere, verificare e provare un unico modulo, che un intero programma.
2. Di solito, uno stesso modulo si rivela utile in varie occasioni, soprattutto se è abbastanza generico e svolge una funzione molto comune. Si potrebbe anche realizzare una libreria di moduli standard.
3. La programmazione modulare consente di separare le varie funzioni e di utilizzare programmi scritti in precedenza.
4. Eventuali elementi da modificare possono essere incorporati in un unico modulo, affinché non sia necessario rivoluzionare l'intero sistema.
5. Gli errori possono essere localizzati meglio e spesso sono limiati ad un singolo modulo.
6. La programmazione modulare ci consente di stabilire meglio i tempi di realizzazione di un progetto, attraverso la valutazione dei risultati intermedi.

SVANTAGGI DELLA PROGRAMMAZIONE MODULARE

L'idea della programmazione modulare è così semplice che spesso gli svantaggi vengono completamente ignorati. Eccone alcuni:

1. Combinare fra loro i vari moduli può essere un compito particolarmente gravoso, soprattutto se sono scritti da persone diverse.
2. I moduli richiedono una documentazione molto dettagliata, in quanto possono agire su parti del programma, come delle strutture dati, utilizzate anche da altri moduli.

Difficoltà di collaudo e di verifica

3. È difficile verificare e collaudare separatamente i vari moduli, dal momento che le loro funzioni sono spesso strettamente correlate. Infatti, alcuni moduli utilizzano gli stessi dati, mentre altri si scambiano i risultati dell'elaborazione. A questo scopo, sarà necessario scrivere dei programmi speciali (chiamati "driver") solo per produrre dei dati campione e collaudare i programmi. Questi driver, però, richiedono un lavoro supplementare, che in definitiva non aggiunge nulla al sistema.

Drivers

Difficoltà di suddivisione

4. La suddivisione di un programma in vari moduli può risultare molto complessa. Del resto, una suddivisione approssimativa finirà per compromettere anche la fase di integrazione poichè quasi tutti i possibili errori e le eventuali modifiche coinvolgeranno parecchi moduli.

5. I programmi modulari comportano un impiego maggiore di tempo e di memoria, poichè moduli diversi ripetono spesso le stesse funzioni.

Quindi, se la programmazione modulare è certamente un miglioramento rispetto al dover scrivere l'intero programma partendo da zero, presenta anche degli svantaggi.

Alcune considerazioni da tenere presenti sono la necessità di limitare il numero delle informazioni comuni a vari moduli, la possibilità di inserire le parti soggette ad eventuali modifiche in un modulo unico e quella di limitare gli scambi di informazioni fra un modulo e l'altro.

PRINCIPI PER LA SUDDIVISIONE IN MODULI

Purtroppo non ci sono dei metodi sistematici e sperimentati per "modularizzare" un programma. Ci limitiamo ad indicare alcuni principi generali:

1. I moduli che utilizzano gli stessi dati dovrebbero far parte di uno stesso modulo più generale.
2. Due moduli, di cui il primo usa il secondo o da esso dipende, ma non viceversa, dovrebbero essere separati.
3. Un modulo utilizzato da parecchi altri moduli deve far parte di un unico modulo globale diverso dagli altri.
4. Due moduli, di cui il primo è utilizzato da molti altri moduli ed il secondo solo da pochi altri, devono restare separati.
5. Due moduli la cui frequenza d'uso è notevolmente diversa dovrebbero rimanere distinti.
6. La struttura, o organizzazione, di dati correlati fra loro dovrà essere racchiusa all'interno di un unico modulo.

Se è difficile rappresentare un programma in forma modulare sarà necessario ridefinire le varie funzioni. Troppi casi speciali o troppe variabili, che richiedono una gestione particolare, testimoniano di una definizione del problema molto approssimativa.

ESEMPI

Rappresentazione Modulare del Sistema Interruttore - Luce

Trattandosi di un programma molto semplice, lo possiamo suddividere in due moduli:

Il Modulo 1 attende che venga premuto l'interruttore e, come risposta, accende la luce.

Il Modulo 2 fornisce un ritardo della durata di un secondo.

Probabilmente il Modulo 1 presenterà delle caratteristiche tutte particolari, dovute al modo in cui sono collegati la luce e l'interruttore. Il Modulo 2 sarà, invece, di utilità più generale, dal momento che sono molte le funzioni che impiegano dei cicli di ritardo. Evidentemente, sarebbe un vantaggio poter disporre di un modulo in grado di fornire intervalli di durata diversa. Nella documentazione, però, dovrà essere indicato come specificare la lunghezza del ritardo, come richiamare il modulo e quali registri e locazioni di memoria sono utilizzati.

Una versione generale del Modulo 1 sarebbe molto meno utile, in quanto dovrebbe operare con differenti tipi di interruttori e di luci, collegati, oltretutto, in modo diverso.

Risulterà, senz'altro, più semplice scrivere un modulo per una configurazione particolare di luci ed interruttori, anzichè cercare di usare una routine standard. È una situazione del tutto diversa da quella del Modulo 2.

Rappresentazione Modulare del Caricatore di Memoria con Interruttori

Non è un compito facile, poichè tutte le funzioni dipendono dalla configurazione dell'hardware e sono estremamente semplici, tanto che i moduli sembrano perfino inutili. Osservando il diagramma di flusso della Figura 17-3, un modulo potrebbe essere quello che attende che l'operatore prema uno dei quattro pulsanti.

Altri moduli potrebbero essere questi:

- Un modulo di ritardo, che fornisce l'intervallo necessario per eliminare i rimbalzi dell'interruttore
- Un modulo interruttore e display, che legge il dato dagli interruttori e lo invia ai display
- Un modulo Lamp Test

I moduli strettamente connessi alle caratteristiche di un particolare sistema, come gli ultimi due, difficilmente ci saranno utili in altre occasioni. Questo non è un caso in cui la programmazione modulare offra grandi vantaggi.

Rappresentazione Modulare di un Terminale di Verifica

Il terminale di verifica, al contrario, si presta ottimamente alla programmazione modulare. L'intero sistema **può essere facilmente suddiviso in tre moduli principali:**

- **Modulo tastiera e display**
- **Modulo trasmissione dati**
- **Modulo ricezione dati**

Sottomoduli del modulo "tastiera e display"

Un modulo tastiera e display sufficientemente generico sarebbe in grado di gestire molti sistemi che impiegano questo tipo di periferiche. I vari sotto-moduli eseguiranno funzioni di questo tipo:

- Riconoscere un nuovo inserimento dalla tastiera e prelevare il dato
- Azzerare la matrice, in risposta al tasto Clear
- Memorizzare le cifre
- Cercare il carattere di fine inserimento o il tasto Send
- Visualizzare le cifre

L'interpretazionie dei tasti ed il numero di cifre varieranno da un caso all'altro, ma la procedura d'inserimento, la memorizzazione dei dati e il processo di visualizzazione saranno gli stessi in molti programmi. Anche certi tasti funzione, come Clear, sono degli standard. Chiaramente, **il progettista deve prestare un'attenzione particolare a quei moduli, che, a suo giudizio, gli saranno utili anche in altre applicazioni.**

Sottomoduli del modulo "trasmissione dati"

Anche il modulo trasmissione dati lo possiamo suddividere in vari sotto-moduli:

1. Aggiungere il carattere d'intestazione.
2 Trasmettere i caratteri, quando la linea di output è in grado di riceverli.
3. Generare un ritardo fra i singoli bit o caratteri.
2. Aggiungere il carattere di coda.
5. Controllare eventuali problemi di trasmissione; cioè, un mancato riconoscimento dei dati o l'impossibilità di trasmettere senza errori.

Sottomoduli del modulo "ricezione dati"

Il modulo ricezione dati comprenderà i sotto-moduli seguenti:

1. Cercare il carattere d'intestazione.
2. Verificare se un messaggio è destinato a quel particolare terminale.
3. Memorizzare ed interpretare il messaggio.
4. Cercare il carattere di coda.
5. Generare un ritardo fra i singoli bit o caratteri.

IL PRINCIPIO DELLE INFORMAZIONI NASCOSTE

A questo punto avrete capito quanto sia importante che ogni soluzione che decidiamo di adottare in fase di progettazione (come la velocità dei bit, il formato di un messaggio o la procedura di rilevamento degli errori) sia poi realizzata all'interno di un solo modulo. Un'eventuale modifica richiederà solo un cambiamento all'interno di quel singolo modulo. Gli altri dovranno restare com-

pletamente all'oscuro di quanto accade all'interno di quel particolare modulo. **È il cosiddetto "principio delle informazioni nascoste"[3], in base al quale i moduli si scambiano solo quelle informazioni che sono assolutamente indispensabili per eseguire la loro funzione. Le altre rimangono nascoste all'interno di ciascun modulo.**

La gestione degli errori è una delle situazioni che richiedono l'applicazione di questo principio. Quando un modulo rileva la presenza di un errore "fatale" non dovrebbe tentare di ripristinare il sistema, ma soltanto informare il modulo chiamante del tipo d'errore, consentendogli di decidere la procedura da seguire. Infatti, un modulo di livello più basso manca di informazioni sufficienti per stabilire le opportune procedure di recupero. Ad esempio, supponiamo che un modulo accetti degli input numerici e si aspetti di ricevere una stringa di cifre chiusa da un ritorno di carrello. L'eventuale inserimento di un carattere non numerico causa un errore. Dal momento che questo modulo non conosce il contesto nel quale è utilizzato (cioè, la stringa numerica è un operando, un numero isolato, il numero di una unità di I/O o la lunghezza di un file?) non è in grado di decidere come gestire l'errore. Se adottasse sempre una stessa procedura di recupero non sarebbe sufficientemente generico e potrebbe essere utilizzato solo in quelle situazioni in cui una tale procedura risulti valida.

REGOLE GENERALI

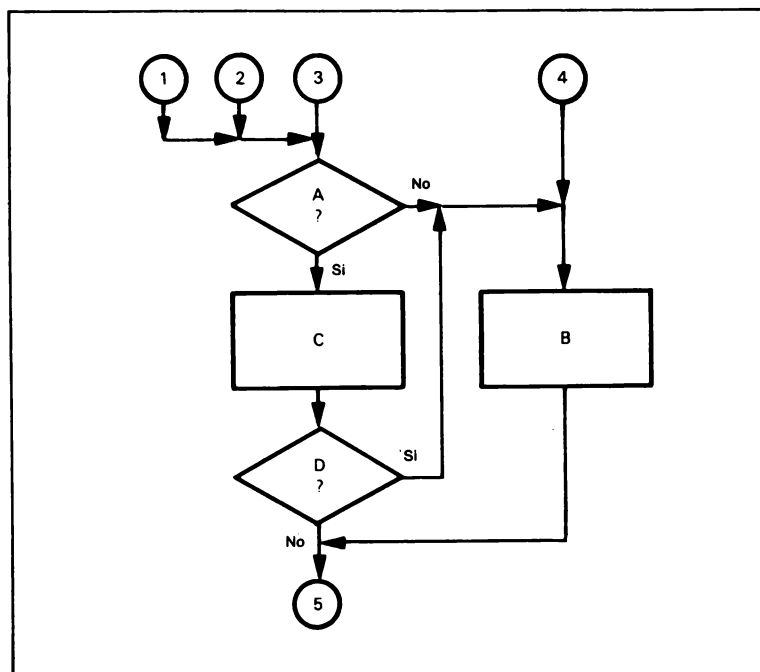
Vi accorgerete che la programmazione modulare è molto utile. È sufficiente attenersi ad alcune regole fondamentali:

Dimensione dei moduli

1. **Utilizzare moduli di lunghezza compresa fra 20 e 50 righe.** Moduli più corti sono, di solito, una perdita di tempo, mentre moduli più lunghi raramente sono abbastanza generici e risultano, quindi, difficili da integrare fra loro.

2. **Realizzare moduli sufficientemente generici.** Distinguere fra caratteristiche comuni alla maggior parte delle applicazioni, come il codice ASCII o i formati per la trasmissione asincrona, e funzioni particolari utilizzate solo in alcuni programmi, come l'identificazione dei tasti, il numero dei dispaly e il numero dei caratteri in un messaggio. La modifica di questi parametri dovrà essere molto semplice. Cambiamenti sostanziali, come l'adozione di un nuovo codice per i caratteri, saranno gestiti all'interno di moduli separati.

3 **Riservare un'attenzione particolare a quei moduli che potranno essere utili anche in altri progetti o che sono utilizzati più volte in uno stesso programma,** come i moduli di ritardo, quelli per la gestione dei dispaly o della tastiera, ecc.

Figura 17-8. Diagramma di Flusso di un Programma Non Strutturato.

4. **Realizzare moduli indipendenti l'uno dall'altro.** Limitare il flusso delle informazioni fra i vari moduli e utilizzare un solo modulo per ciascuna funzione.
5. **Non suddividere in moduli funzioni che sono già molto semplici e facilmente realizzabili.**

PROGRAMMAZIONE STRUTTURATA

Come mantenere distinti i vari moduli ed impedire interazioni reciproche? In che modo scrivere un programma che abbia una sequenza di operazioni sufficientemente chiara, così da poter meglio individuare e correggere eventuali errori? Una soluzione è quella di usare la cosiddetta "programmazione strutturata", un metodo in cui ogni parte del programma è formata da elementi appartenenti ad un set ben definito di strutture, provviste di un unico ingresso e di un'unica uscita. La Figura 17-8 mostra un diagramma di flusso di un programma non strutturato. Se si verifca un errore nel Modulo B, sono cinque le cause possibili. Non solo dovremo controllare ciascuna sequenza, ma anche assicurarci che le modifiche apportate ad una sequenza non abbiano effetti negativi su nessuna delle altre. Fare il debugging di un programma come questo è come lottare con una piovra. Ogni volta che crediamo di avere la situazione sotto controllo, c'è un altro tentacolo libero da un'altra parte.

La soluzione è quella di **definire una sequenza molto chiara delle varie operazioni, che permetta di localizzare meglio gli errori ed utilizzi strutture ad un solo ingresso e ad una sola uscita.** Un programma risulterà formato da una sequenza di strutture; può trattarsi di un singolo statement oppure di strutture nidificate l'una nell'altra, fino ad un qualsiasi livello di complessità. **Qui di seguito sono elencate le strutture necessarie.**

Strutture di sequenziazione

1. **Una sequenza ordinaria;** cioè, una struttura lineare in cui i programmi sono eseguiti di seguito l'uno all'altro. Se la sequenza è

   P1
   P2
   P3

   L'elaboratore esegue prima P1, poi P2 ed infine P3. P1, P2 e P3 possono essere istruzioni singole o programmi molto complessi.

Strutture di selezione

2. **Una struttura condizionale, in cui l'esecuzione di un programma dipende da una certa condizione.**
   Sono molte le varianti possibili, ma una piuttosto comune è "if C then P1 else P2" (se C allora P1 altrimenti P2), dove C è una condizione, mentre P1 e P2 sono dei programmi. Il computer esegue P1 se C è vero, P2 se C è falso. La Figura 17-9 mostra la logica di questa struttura. Essa ha un solo ingresso ed una sola uscita: non sono possibili alternative diverse.

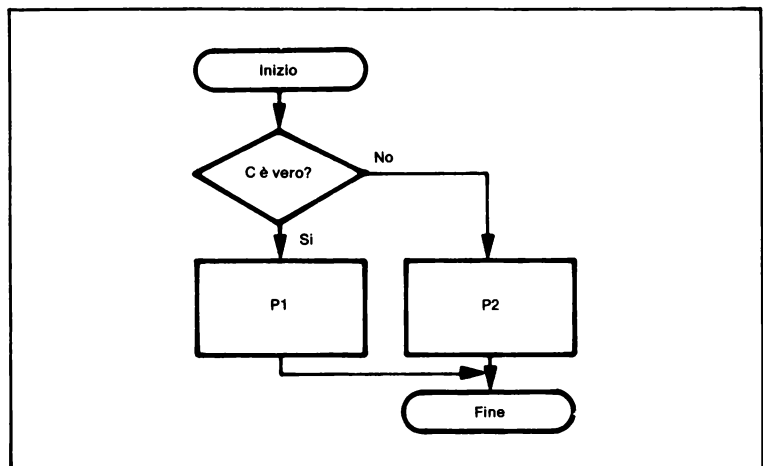



Figura 17-9. Diagramma di Flusso della Struttura If-Then-Else

Strutture di iterazione

3. **Una struttura ciclica, in cui un programma viene ripetuto finchè (o per tutto il tempo che) una condizione è valida.**
Ci sono varie possibilità. Una piuttosto comune (chiamata struttura "do-while") è "while C do P" (per tutto il tempo in cui C è verificata esegui P), dove C è una condizione e P un programma. L'elaboratore controlla continuamente C e, quindi, esegue P finchè la condizione C è vera.
Un 'alternativa ovvia è "until C do P" (fino a che non C esegui P), in cui il computer controlla continuamente C ed esegue P per tutto il tempo in cui C è falso. Le Figure 17-10 e 17-11 mostrano la logica di queste strutture. Entrambe hanno un solo ingresso ed una sola uscita. Il computer non eseguirà affatto P, se C è fin dall'inizio nello stato di uscita; perciò, P non è automaticamente eseguito almeno una volta, come avviene, invece, nel ciclo DO del FORTRAN oppure nel caso di strutture alternative, come "do P while C" (esegui P per tutto il tempo in cui C), o "repeat P until C" (ripeti P fino a che non C), in cui il computer controlla la condizione solo dopo aver eseguito il

Figura 17-10. Diagramma di Flusso della Struttura Do-While

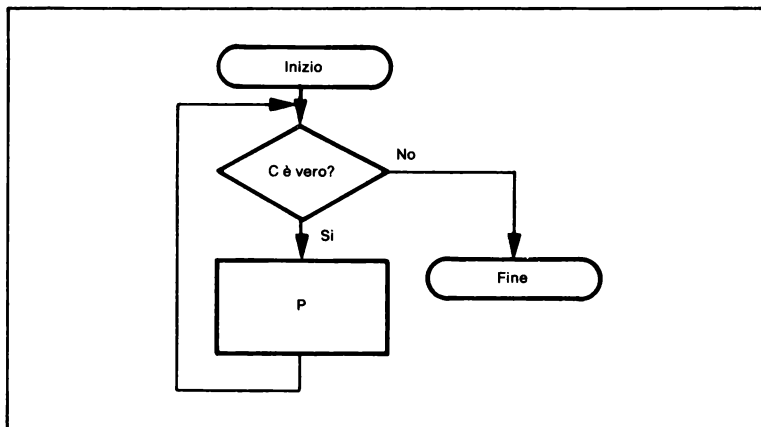
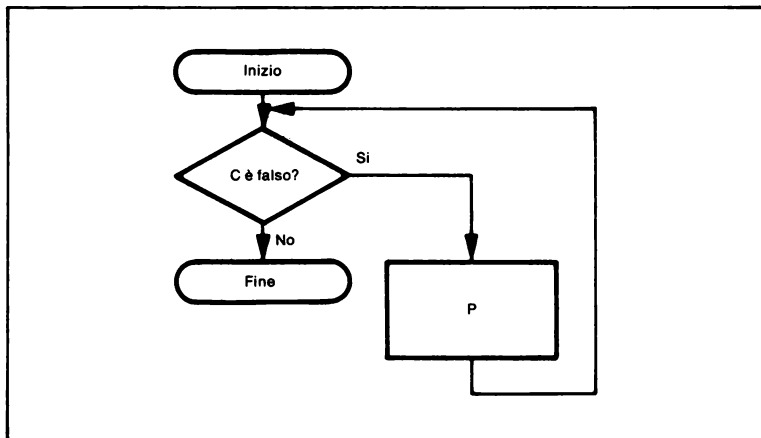



Figura 17-11. Diagramma di Flusso della struttura Do-Until.

programma (ricordate le Figure 5-1 e 5-2). Questo metodo spesso si rivela più efficace, ma noi utilizzeremo solo la forma della Figura 17-10 per semplificare la discussione. La maggior parte dei linguaggi strutturati permettono di usare tutte e quattro le alternative, in modo da garantire una certa flessibilità. Nella maggioranza dei casi, è il programma P che, alla fine, mette C nello stato che provoca l'uscita; se non lo fa, l'elaboratore continuerà ad eseguire P all'infinito (la cosiddetta struttura DO FOREVER); come , del resto, accadrà nel caso che P sia un programma per il controllo di un'apparecchiatura, di una periferica, di un sistema di collaudo o di un videogame.

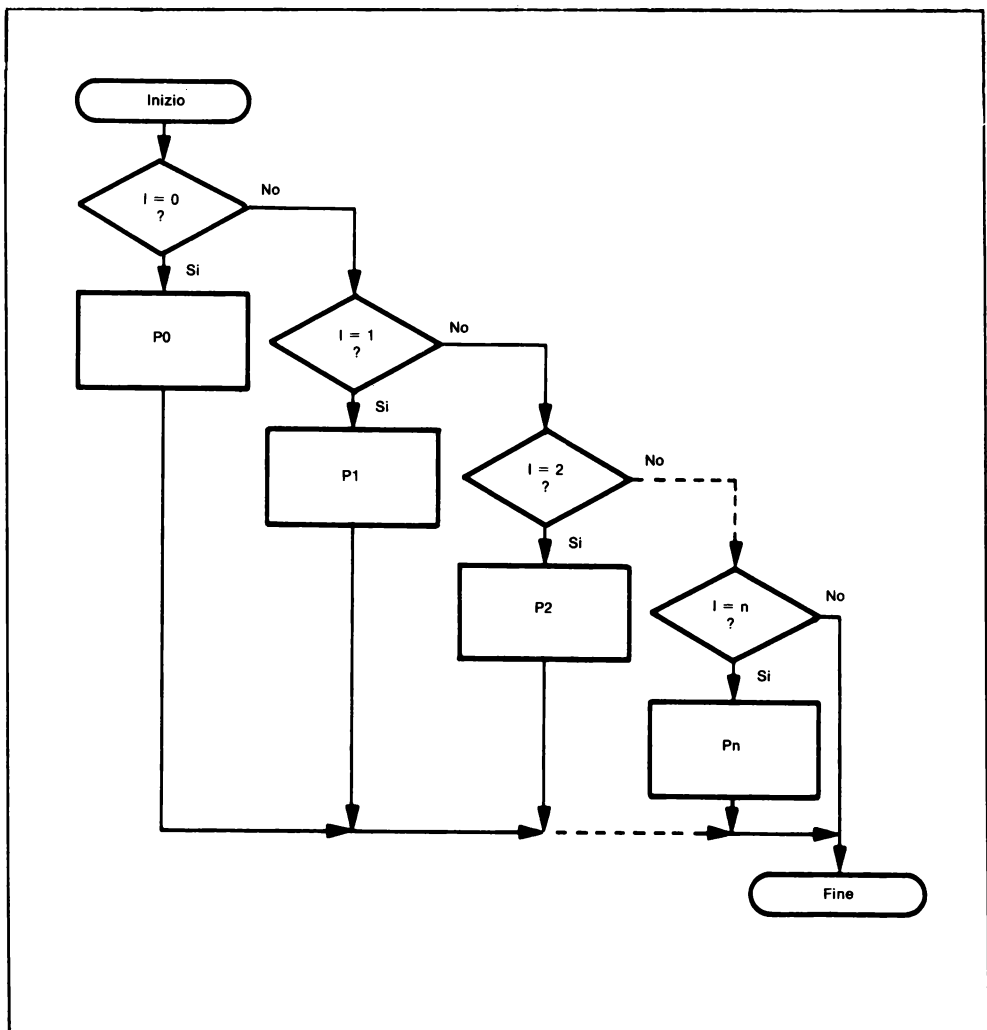



Figura 17-12. Diagramma di flusso di una struttura Case.

Struttura di selezione "case"

4. **La struttura "case".** Sebbene non si tratti di una struttura fondamentale come le prime tre, è così diffusa che merita una descrizione a parte. È indicata con "case I of P0, P1,.....,Pn", dove I è un indice e P0, P1,....., Pn sono programmi. Un computer esegue il programma P0 se I è 0, P1 se I è 1, e così via; sarà eseguito soltanto uno degli n programmi. Dopo l'esecuzione di uno dei programmi oppure se I è maggiore di n (il numero di programmi disponibili) il processore, eseguirà l'istruzione successiva, come è indicato nella Figura 17-12. Naturalmente, potremmo sostituire una struttura "case" con una serie di strutture condizionali, allo stesso modo in cui una tabella di salto (jump table) potrebbe essere sostituita da una serie di diramazioni condizionate. Non si tratterebbe, comunque, di una soluzione molto elegante ed altrettanto efficace.

CARATTERISTICHE ED ESEMPI DELLE STRUTTURE

Ecco alcune caratteristiche della programmazione strutturata:

1. **Sono consentite soltanto le tre strutture fondamentali ed un piccolo numero di strutture ausiliare.** Sono possibili inoltre delle varianti dei loop e delle strutture condizionali.
2. **Le strutture possono essere nidificate fino ad un livello qualsiasi,** dal momento che ogni struttura può, a sua volta, contenere altre strutture.
3. **Ogni struttura ha un solo ingresso ed una sola uscita.**

Questi sono alcuni esempi della struttura condizionale illustrata nella Figura 17-9:

1. P2 presente:

```
IF X > 0 THEN NPOS = NPOS + 1
        ELSE NNEG = NNEG + 1
```

Sia P1 che P2 sono costituiti da singoli statement

2. P2 omesso:

```
IF X = 0 THEN Y = 1/X
```

In questo caso, se C è falso (X = 0), non succede niente. P2 ed "else" possono anche essere omessi. **Questi sono alcuni esempi della struttura ciclica mostrata nella Figura 17-10:**

1. Eseguire la somma degli interi da 1 a N

```
I = 0
SUM = 0
DO WHILE I < N
      I = I + 1
      SUM = SUM + I
END
```

Il processore esegue il ciclo per tutto il tempo in cui I < N. Se N = 0 il programma all'interno della struttura "do-while" non viene eseguito affatto.

2. Contare i caratteri nel vettore SENTENCE, finchè non si trova il codice ASCII corrispondente al punto.

```
NCHAR = 0
DO WHILE SENTENCE(NCHAR) < > PERIOD
      NCHAR = NCHAR + 1
END
```

Il computer esegue il loop finchè non incontra un punto. Il risultato sarà zero se il primo carattere è un punto.

VANTAGGI DELLA PROGRAMMAZIONE STRUTTURATA

I vantaggi della programmazione strutturata sono:

1. È semplice seguire il succedersi delle varie operazioni. Questo permette di collaudare e verificare il programma con molta facilità.
2. Il numero delle strutture è limitato e la terminologia standardizzata.
3. Le strutture sono adatte ad una rappresentazione modulare.
4. Studi teorici hanno dimostrato che le strutture attualmente disponibili sono sufficienti; con esse è possibile scrivere qualsiasi tipo di programma.
5. La versione strutturata di un programma si documenta da sola ed è abbastanza comprensibile.
6. È facile individuare e descrivere le caratteristiche fondamentali di un programma strutturato.
7. È stato dimostrato, in modo empirico, che la programmazione strutturata aumenta notevolmente la produttività di un programmatore.

In sostanza, la programmazione strutturata costringe un programmatore ad una maggiore rigorosità, rispetto alla programmazione modulare. Ne risultano programmi più razionali ed efficienti.

SVANTAGGI DELLA PROGRAMMAZIONE STRUTTURATA

Gli svantaggi della programmazione strutturata sono:

1. Solo alcuni linguaggi ad alto livello (ad es. PL/M, Pascal) accettano le strutture. Un programmatore, quindi, deve passare attraverso un'ulteriore fase di traduzione, per convertirle in linguaggio assembly. Tuttavia, la versione strutturata del programma è utile come documentazione.
2. I programmi strutturati sono spesso più lenti e utilizzano più memoria di quelli non strutturati.
3. Disporre di sole tre strutture fondamentali rende piuttosto complesso eseguire certe funzioni. Con esse possiamo realizzare qualsiasi tipo di programma, ma non è detto che ciò sia molto semplice e che si ottengano sempre dei buoni risultati.
4. Le strutture standard creano spesso confusione: ad es., è molto difficile leggere strutture nidificate di tipo "if-then-else", dato che manca il modo per indicare dove terminano le strutture più interne. Anche una serie di loop "do-while" nidificati risulta a volte incomprensibile.
5. I programmi strutturati tengono conto soltanto della sequenza delle operazioni, ma non del flusso dei dati. Perciò, le strutture finiscono per gestire i dati in modo inadeguato.
6. Sono pochi i programmatori abituati alla programmazione strutturata. Sono molti, invece, a trovare le strutture standard scomode e restrittive.

QUANDO USARE LA PROGRAMMAZIONE STRUTTURATA

Non abbiamo intenzione nè di favorire, nè di scoraggiare l'impiego della programmazione strutturata. In linea generale, essa si rivela molto più utile nei seguenti casi:

- Programmi molto lunghi (oltre 1000 istruzioni).
- Applicazioni in cui la quantità di memoria utilizzata non è importante.
- Applicazioni in cui i costi di sviluppo del software, in particolare il collaudo e la verifica, incidono in maniera significativa.
- Applicazioni che prevedono la manipolazione di stringhe, un controllo di processo o altri algoritmi, anzichè una semplice manipolazione di bit.

Tendenze future

In futuro, il costo della memoria diminuirà, mentre aumenteranno, in media, le dimensioni dei programmi ed i costi di sviluppo del

software. Questo depone a favore della programmazione strutturata, che, soprattutto nel caso di programmi molto lunghi, riduce l'incidenza dei costi di sviluppo, molto spesso a scapito di un'efficiente utilizzazione della memoria.

Programmazione Strutturata e Linguaggio Assembly

Solo perchè i concetti della programmazione strutturata sono espressi generalmente in linguaggi ad alto livello, non significa che essa non sia applicabile al linguaggio assembly. Al contrario, **il programmatore in linguaggio assembly, con la totale libertà di espressione che questo linguaggio gli consente, ha bisogno di un certo grado di strutturazione, che solo la programmazione strutturata può offrirgli. La creazione di moduli con dei singoli punti di ingresso e di uscita, l'uso di semplici strutture di controllo e la possibilità di ridurre la complessità di ciascun modulo accrescono notevolmente la produttività di un programmatore in linguaggio assembly.**

ESEMPI

Programma Strutturato per il Sistema Interruttore-Luce

La versione strutturata di questo esempio è la seguente:

```
SWITCH = OFF
DO WHILE SWITCH = OFF
      READ SWITCH
      END
LIGHT = ON
DELAY 1
LIGHT = OFF
```

ON ed OFF devono essere definiti in modo adeguato per l'interruttore e la luce. DELAY è un modulo che fornisce un ritardo di una certa durata, in base al relativo parametro espresso in secondi.

Uno statement in un programma strutturato corrisponde, in alcuni casi, ad una subroutine, che, dovendosi conformare alle regole della programmazione strutturata, non può far altro, alla fine, che restituire il controllo al programma principale.

Dal momento che "do-while" controlla la validità della condizione prima di eseguire il loop, prima di cominciare poniamo ad OFF la variabile SWITCH. Il programma strutturato è immediato, comprensibile e facile da controllare manualmente. Richiederà, tuttavia, una disponibilità di memoria maggiore rispetto ad un programma non strutturato, che non avrebbe da inizializzare SWITCH e potrebbe fondere le procedure di lettura e di controllo.

Programma Strutturato per il Caricatore di Memoria con Interruttori

Il caricatore di memoria con interruttori costituisce un problema di

programmazione strutturata ben più complesso. Partendo dal diagramma di flusso della Figura 17-3, otterremo un programma di questo tipo (con l'asterisco indichiamo un commento, mentre utilizziamo "begin" ed "end" rispettivamente all'inizio ed alla fine di un programma formato da più di una riga ed eseguito solo nel caso che si verfichi una certa condizione):

```
*
*AZZERA L'IND. ALL'INIZIO COSÌ IL VALORE DI PARTENZA È ZERO
*
HIADDRESS = 0
MIDADDRESS = 0
LOADDRESS = 0
*
*ESAMINA SEMPRE GLI INTERRUTTORI E CARICA I DATI IN MEMORIA
*   SI NOTI CHE "DO FOREVER" È UN "DO WHILE" SENZA CONDIZIONE
*
DO FOREVER
*
*CONTROLLA PULSANTE IND. ALTO. SE PREMUTO, ELIMINA RIMBALZO
*   E ATTENDI CHE SIA RILASCIATO. QUINDI LEGGI INDIR. ALTO
*   DAGLI INTERRUTTORI E MOSTRALO SUI DISPLAY
*
    IF HIGHADDRBUTTON = 0 THEN
        BEGIN
            DO WHILE HIADDRBUTTON = 0
                DELAY (DEBOUNCE TIME)
                END
            HIADDRESS = SWITCHES
            LIGHTS = SWITCHES
        END


*CONTROLLA PULSANTE IND. MEDIO. SE PREMUTO, ELIMINA RIMBALZO
*   E ATTENDI CHE SIA RILASCIATO. QUINDI LEGGI INDIR. MEDIO
*   DAGLI INTERRUTTORI E MOSTRALO SUI DISPLAY
*
    IF MIDADDRBUTTON = 0 THEN
        BEGIN
            DO WHILE MIDADDRBUTTON = 0
                DELAY (DEBOUNCE TIME)
                END
            MIDADDRESS = SWITCHES
            LIGHTS = SWITCHES
        END


*CONTROLLA PULSANTE IND. BASSO. SE PREMUTO, ELIMINA RIMBALZO
*   E ATTENDI CHE SIA RILASCIATO. QUINDI LEGGI INDIR. BASSO
*   DAGLI INTERRUTTORI E MOSTRALO SUI DISPLAY
*
    IF LOADDRBUTTON = 0 THEN
        BEGIN
            DO WHILE LOADDRBUTTON = 0
                DELAY (DEBOUNCE TIME)
                END
            LOADDRESS = SWITCHES
            LIGHTS = SWITCHES
        END
```

```
*CONTROLLA PULSANTE DATO. SE PREMUTO, ELIMINA RIMBALZO
*   E ATTENDI CHE SIA RILASCIATO. QUINDI LEGGI DATO
*   DAGLI INTERRUTTORI E MOSTRALO SUI DISPLAY E SALVALO IN MEMORIA
*   A (IND. ALTO, IND. MEDIO, IND. BASSO)
*
    IF DATABUTTON = 0 THEN
        BEGIN
            DO WHILE DATABUTTON = 0
                DELAY (DEBOUNCE TIME)
                END
            DATA= SWITCHES
            LIGHTS = SWITCHES
            (HIADDRESS, MIDADDRESS, LOADDRESS) = DATA
        END

*
*ATTENDI IL TEMPO NECESSARIO PER ELIMINARE I RIMBALZI PRIMA DI
*   ESAMINARE DI NUOVO I PULSANTI. QUESTO INTERVALLO ELIMINA
*   I RIMBALZI QUANDO UN PULSANTE VIENE RILASCIATO
*
    DELAY (DEBOUNCE TIME)
END
*
*L'ULTIMO END CHIUDE IL
*  CICLO DO FOREVER
*
```

Non è facile scrivere un programma strutturato, ma con esso abbiamo un quadro completo della logica complessiva del programma, potendo così verificarne l'esattezza prima di scrivere il codice vero e proprio.

Programma strutturato per il Terminale di Verifica

Prendiamo in esame l'inserimento da tastiera per il terminale di transazione. ENTRY è il vettore del display, KEYSTROBE è il segnale di strobe e KEYIN il dato della tastiera. **Il programma strutturato, senza i tasti funzione, è il seguente:**

```
NKEYS = 10
*
*AZZERA ENTRY PRIMA DI INIZIARE
*
    DO WHILE NKEYS > 0
        NKEYS = NKEYS - 1
        ENTRY (NKEYS) = 0
    END

*
*PRELEVA UN ELEMENTO COMPLETO DALLA TASTIERA
*
    DO WHILE NKEYS < 10
        IF KEYSTROBE = ACTIVE THEN
            BEGIN
```

```
                KEYSTROBE = INACTIVE
                ENTRY (NKEYS) = KEYIN
                NKEYS = NKEYS + 1
            END
        END
```

L'aggiunta del tasto SEND costringe il programma ad ignorare eventuali cifre in più inviate dopo un inserimento completo ed a trascurare il tasto SEND, finchè non è terminato un inserimento. Il relativo programma strutturato è il seguente:

```
NKEYS = 10
*
*AZZERA ENTRY PRIMA DI INIZIARE
*
   DO WHILE NKEYS > 0
        NKEYS = NKEYS - 1
        ENTRY (NKEYS) = 0
   END
*

*ATTENDI UN INSERIMENTO COMPLETO SEGUITO DAL TASTO SEND
*
   DO WHILE KEY SEND OR NKEYS 10
        IF KEYSTROBE = ACTIVE THEN
          BEGIN
              KEYSTROBE = INACTIVE
              KEY = KEYIN
              IF NKEYS 10 AND KEY SEND THEN
                BEGIN
                  ENTRY (NKEYS) = KEYIN
                  NKEYS = NKEYS + 1
                END
          END
   END
```

Notate le caratteristiche di questo programma

1. Il secondo if-then è nidificato all'interno del primo, in quanto i tasti sono accettati solo dopo il riconoscimento del segnale di strobe. Se il secondo if-then fosse allo stesso livello del primo, un solo tasto sarebbe sufficiente a completare l'inserimento, poichè il suo valore verrebbe immesso nel vettore ad ogni iterazione del loop do-while.
2. KEY non deve essere definita all'inizio, dal momento che NKEYS viene posta a zero durante l'inizializzazione di ENTRY.

L'aggiunta del tasto CLEAR consente al programma di azzerare, inizialmente, ENTRY, simulando la pressione di CLEAR, ponendo,

cioè, NKEYS uguale a 10 e KEY uguale a CLEAR prima di cominciare. Il programma deve anche azzerare le cifre che sono state introdotte precedentemente. **Il nuovo programma sarà:**

```
*
* SIMULARE UN COMPLETO AZZERAMENTO
*
NKEYS = 10
KEY = CLEAR
*
*ATTENDI UN INSERIMENTO COMPLETO ED IL TASTO SEND
*
DO WHILE KEY = SEND OR NKEYS = 10
*
*AZZERA ENTRY SE È STATO PREMUTO IL TASTO CLEAR
*
    IF KEY = CLEAR THEN
    BEGIN
        KEY = 0
        DO WHILE NKEYS > 0
           NKEYS = NKEYS - 1
           ENTRY(NKEYS) = 0
        END
    END

 *
 *PRENDI CIFRA SE L'INSERIMENTO NON È COMPLETO
         *

          IF KEYSTROBE = ACTIVE THEN
            BEGIN
                KEYSTROBE = INACTIVE
                IF KEY < 10 AND NKEY 10 THEN
                  BEGIN
                     ENTRY (NKEYS) = KEY
                     NKEYS = NKEYS + 1
                  END
            END
     END
```

Il programma pone KEY uguale a zero dopo aver azzerato il vettore, in modo che l'operazione non venga ripetuta.

Possiamo anche realizzare un programma strutturato per la routine di ricezione. Inizialmente ci limiteremo a verificare la presenza dei caratteri d'intestazione e di coda. RSTB sta ad indicare la disponibilità di un carattere. **Il relativo programma strutturato è il seguente:**

```
*
*AZZERA IL FLAG D'INTESTAZIONE PRIMA DI INIZIARE
*
HFLAG = 0
*
*ATTENDI L'INTESTAZIONE E LA CODA
*
DO WHILE HFLAG = 0 OR CHAR TRAILER
*
```

```
*PRENDI UN CARATTERE SE È DISPONIBILE. CERCA L'INTESTAZIONE
*
    IF RSTB = ACTIVE THEN
      BEGIN
          RSTB = INACTIVE
          CHAR = INPUT
          IF CHAR = HEADER THEN HFLAF = 1
      END
END
```

Adesso possiamo aggiungere la sezione che confronta l'indirizzo del messaggio con le tre cifre dell'indirizzo del terminale (TERMADDR). Se anche una sola delle tre cifre non corrisponde, il relativo flag (ADDRMATCH) è messo a 1.

```
*
*PRIMA DI INIZIARE, AZZERA FLAG D'INTESTAZIONE, DI UGUAGLIANZA
*   E CONTATORE INDIRIZZO
*
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
*
*ATTENDI L'INTESTAZIONE, L'IND. DI DESTINAZIONE E LA CODA
*
DO WHILE HFLAG = 0 OR CHAR = TRAILER OR ADDRCTR = 3
*
*PRENDI UN CARATTERE SE È DISPONIBILE
*
    IF RSTB = ACTIVE THEN
        BEGIN
            RSTB = INACTIVE
            CHAR = INPUT
        END


*
*CONTROLLA L'INDIRIZZO DEL TERMINALE E L'INTESTAZIONE
*
    IF HFLAG = 1 AND ADDRCTR = 3 THEN
        BEGIN
            IF CHAR = TERMADDR(ADDRCTR) THEN ADDRMATCH = 1
            ADDRCTR = ADDRCTR+1
        END
    IF CHAR = HEADER THEN HFLAG = 1
END
```

Adesso il programma attende un'intestazione, un codice di identificazione a tre cifre ed un carattere di coda. Bisogna fare attenzione a ciò che accade quando il programma trova l'intestazione e quando, per errore, un carattere del codice di identificazione è uguale al carattere di coda.

Inoltre, possiamo salvare il messaggio nel vettore MESSG, con NMESS che indica il numero di caratteri presenti; se, alla fine, il suo

valore è diverso da zero, il programma sa che il terminale ha ricevuto un messaggio valido. In questo caso, non ci siamo preoccupati di ridurre le espressioni logiche.

```
*
*PRIMA DI INIZIARE, AZZERA I FLAG ED I CONTATORI
*
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
NMESS = 0
*
*ATTENDI L'INTESTAZIONE, L'IND. DI DESTINAZIONE E LA CODA
*
DO WHILE HFLAG = 0 OR CHAR = TRAILER OR ADDRCTR = 3
*
*PRENDI UN CARATTERE SE È DISPONIBILE
*
   IF RSTB = ACTIVE THEN
      BEGIN
         RSTB = INACTIVE
         CHAR = INPUT
      END

*
*LEGGI IL MESSAGGIO SE IND. DESTINAZIONE = IND. TERMINALE
*
   IF HFLAG = 1 AND ADDRCTR = 3 THEN
      IF ADDRMATCH = 0 AND CHAR TRAILER THEN
         BEGIN
            MESSG(NMESS) = CHAR
            NMESS = NESS + 1
         END

*
*CONTROLLA L'INDIRIZZO DEL TERMINALE
*
   IF HFLAG = 1 AND ADDRCTR 3 THEN
      BEGIN
         IF CHAR = TERMADDR(ADDRCTR) THEN ADDRMATCH = 1
         ADDRCTR = ADDRCTR + 1
      END

*
*CONTROLLA L'INTESTAZIONE
*
   IF CHAR HEADER THEN HFLAG = 1
END
```

Il programma verifica la presenza del codice di identificazione solo se ha trovato un'intestazione durante l'iterazione precedente. Accetta il messaggio soltanto se ha già trovato un'intestazione ed un indirizzo di destinazione completo e valido. Il programma deve funzionare correttamente durante le iterazioni, quando trova l'intestazione, la coda e l'ultima cifra dell'indirizzo di destinazione. Non deve cercare di confrontare l'intestazione con l'indirizzo del terminale o di considerare come parte del messaggio il carattere di coda o l'ultima cifra dell'indirizzo di destinazione. **Potete provare ad ag-**

giungere al programma il resto della logica, aiutandovi con il relativo diagramma di flusso (Figura 17-7). Non dimenticate che l'ordine delle operazioni è spesso critico: è necessario accertarsi che il programma non completi una fase e dia inizio a quella successiva nell'ambito di una stessa iterazione.

CONCLUSIONI

La programmazione strutturata garantisce un certo ordine nella stesura di un programma. Costringe ad usare un numero limitato di strutture e a ridurre la sequenza delle operazioni. Fornisce strutture a singola entrata e singola uscita, delle quali è possibile controllare la correttezza logica. La programmazione strutturata consente al progettista di notare eventuali incoerenze o delle possibili combinazioni degli input. Insomma non è una panacea, ma serve a portare ordine in un processo, che, altrimenti, potrebbe risultare caotico. Si dimostra utile anche nelle fasi di debugging, collaudo e documentazione.

La programmazione strutturata non è semplice. Il programmatore non solo deve definire il problema in modo adeguato, ma deve anche cercare di migliorarne la logica. È una procedura spesso noiosa e difficile, ma vi permetterà di ottenere un programma ben scritto e funzionante.

Terminatori

Le strutture che abbiamo descritto non sono le migliori in assoluto, anzi spesso risultano troppo farraginose. Inoltre, a volte è difficile stabilire dove finisce una struttura e ne comincia un'altra, soprattutto se si tratta di strutture nidificate. In futuro i teorici riusciranno senz'altro a darci strutture migliori oppure i progettisti ne aggiungeranno di proprie. Un "terminatore" per ogni struttura sembra indispensabile, poichè variando l'indentazione non si ottengono grandi risultati in termini di chiarezza. "End" è un ovvio terminatore per un ciclo "do-while". Per lo statement "if-then-else", tuttavia, non c'è un terminatore scontato; alcuni hanno suggerito "endif" oppure "fi" ("if" rovesciato), ma non sembrano delle soluzioni molto brillanti e la leggibilità del programma non ne guadagna.

REGOLE DELLA PROGRAMMAZIONE STRUTTURATA

Ecco le regole che vi consigliamo di seguire:

1. **Iniziare con un diagramma di flusso molto generale,** per meglio definire la logica del programma.

2. **Cominciare dalle strutture "sequenziali", "if-then-else" e "do-while".** Esse costituiscono un set di strutture completo, sono sufficienti, cioè, per qualunque tipo di programma.
3. **Usare una indentazione diversa** per ciascun livello, aumentandola rispetto al livello precedente, in modo da riconoscere quali statement ne fanno parte.
4. **Usare dei terminatori per ogni struttura:** ad es., "end" per la struttura "do-while", "endif" o "fi" per quella "if-then-else". I terminatori, insieme alla indentazione, dovrebbero rendere il programma abbastanza chiaro.
5. **Cercare di ottenere il massimo di semplicità e leggibilità.** Lasciare molti spazi, usare nomi che abbiano un certo significato ed espressioni molto chiare. Non ridurre le espressioni logiche, se questo va a danno della chiarezza.
6. **Commentare il programma** in modo sistematico.
7. **Verificare la logica.** Provare tutti i casi estremi o le situazioni particolari ed eventualmente alcuni casi più semplici. Ogni errore individuato in questa fase non causerà dei problemi in seguito.

PROGETTAZIONE TOP-DOWN

Rimane il problema di come controllare ed integrare fra loro i moduli e le strutture. Dovremo senz'altro suddividere un compito molto vasto in vari sottocompiti; ma in che modo collaudarli separatamente e metterli, poi, tutti insieme? La procedura standard, chiamata "progettazione bottom-up" (dal basso in alto), richiede tempi più lunghi per il collaudo e la verifica, lasciando l'intero processo di integrazione alla fine. Ciò di cui abbiamo bisogno è un metodo che consenta il collaudo e la verifica nel contesto reale del programma e renda modulare l'integrazione del sistema.

Procedura da seguire nella progettazione Top-Down

È la cosiddetta "progettazione top-down" (dall'alto in basso). Si comincia con lo scrivere la parte principale del programma. Si sostituiscono i sottoprogrammi con dei "monconi" (stub), programmi temporanei che memorizzano l'ingresso, forniscono una risposta ad un problema usato come test oppure non fanno assolutamente niente. Quindi, effettuiamo un controllo per verificare se la logica è corretta.

Passiamo ad espandere i "monconi", ciascuno dei quali conterrà dei sottocompiti che, ancora una volta, rappresenteremo in forma di "monconi". Questo processo di espansione, verifica e collaudo proseguirà finchè tutti i "monconi" non saranno stati sostituiti con programmi funzionanti. Si noti come il collaudo e l'integrazione avvengano ad ogni livello, invece di essere rimandati alla parte finale. Non sono necessari nè dei driver speciali, nè programmi per generare i dati e, inoltre, sappiamo sempre con esattezza a quale punto del progetto ci troviamo. **La progettazione top-down presuppone l'impiego della programmazione modulare ed è compatibile con la programmazione strutturata.**

SVANTAGGI DELLA PROGETTAZIONE TOP-DOWN

Gli svantaggi connessi alla progettazione top-down sono:

1. Il progetto complessivo può non adattarsi bene all'hardware utilizzato.
2. Non sfrutta al meglio il software già esistente.
3. È difficile scrivere dei "monconi", soprattutto se devono funzionare correttamente in parecchi punti diversi.
4. La progettazione top-down spesso non fornisce dei moduli abbastanza generici da poter essere riutilizzati.
5. Eventuali errori al livello più alto hanno spesso degli effetti catastrofici, mentre, nel caso della progettazione bottom-up, gli errori restano, in genere, circoscritti ad un singolo modulo.

Produttività nello sviluppo Top-Down

Nel caso di programmi molto lunghi, la progettazione top-down si è dimostrata in grado di aumentare notevolmente la produttività del programmatore. Tuttavia, anche in occasioni come queste, si preferisce la tecnica bottom-up ogni qualvolta il metodo top-down richieda parecchio lavoro in più.

La progettazione top-down è uno strumento molto utile, ma non dovrebbe essere impiegata in modo esasperato. Garantisce alle fasi di collaudo ed integrazione lo stesso livello di ordine e chiarezza, che la programmazione strutturata fornisce alla progettazione dei moduli, oltre ad avere un campo di applicazione molto più vasto, dal momento che non presuppone l'impiego di logica programmata. Tuttavia, non sempre si ottengono dei buoni risultati in termini di efficienza.

ESEMPI

Progettazione Top-Down del Sistema Interruttore-Luce

Il primo esempio di programmazione strutturata, in realtà, forniva anche una dimostrazione di progettazione top-down. Il programma era:

```
SWITCH = OFF
DO WHILE SWITCH = OFF
     READ SWITCH
END
LIGHT = ON
DELAY 1
LIGHT = OFF
```

Questi statement sono, in realtà, dei "monconi", in quanto nessuno di essi è completamente definito. Ad esempio, cosa significa READ SWITCH? Se l'interruttore fosse un bit della porta di input SPORT, allora significherebbe:

```
SWITCH = SPORT AND SMASK
```

dove SMASK ha un bit '1' nella posizione giusta. Naturalmente, il mascheramento può essere ottenuto, poi, con un'istruzione Bit Test. Analogamente, DELAY 1 significa (nel caso sia il processore stesso a produrre il ritardo):

```
REG = COUNT
DO WHILE REG < > 0
     REG = REG - 1
END
```

COUNT è il valore corrispondente ad un ritardo di un secondo. **La versione espansa del programma è:**

```
SWITCH = 0
DO WHILE SWITCH = 0
     SWITCH = SPORT AND MASK
END
LIGHT = ON
REF = COUNT
DO WHILE REG = 0
     REG = REG - 1
END
LIGHT - NOT(LIGHT)
```

Questo è, senz'altro, un programma più chiaro e potrebbe essere facilmente tradotto in istruzioni o statement reali.

Progettazione Top-Down del Caricatore di memoria con Interruttori

Questo esempio è più complesso di quello precedente, perciò dobbiamo procedere in modo sistematico. Anche in questo caso **utilizzeremo dei "monconi".**

Ad esempio, se il pulsante HIGH ADDRESS è un bit della porta di input CPORT, "if HIADDRBUTTON = 0" significherà:

1. Input dalla porta CPORT
2. AND logico con HAMASK

dove HAMASK ha un '1' nel bit opportuno e '0' negli altri. Analogamente, la condizione "if DATABUTTON = 0" in realtà significa:

1. Input da CPORT
2. AND logico con DAMASK

Così i "monconi" iniziali si limitano a supporre che non sia stato premuto nessun pulsante:

```
HIADDRBUTTON = 1
MIDADDRBUTTON = 1
LOADDRBUTTON = 1
DATABUTTON = 1
```

L'esecuzione del programma supervisore renderà evidente che esso segue il percorso implicito "else" nelle strutture "if-then-else" e non legge mai gli interruttori. Analogamente, se il "moncone" fosse:

```
HIADDRBUTTON = 0
```

il programma dovrebbe rimanere nel loop "do while HIADDRBUTTON = 0", aspettando che il pulsante venga rilasciato. Queste semplici esecuzioni servono a controllare la logica complessiva.

Adesso possiamo espandere ogni "moncone" e verificare se l'espansione produce un risultato complessivo valido. Notate come la verifica ed il collaudo avvengano automaticamente, modulo per modulo. Espanderemo il modulo HIADDRBUTTON = 0 in:

```
READ CPORT
HIADDRBUTTON = (CPORT) AND HAMASK
```

Il programma attende che il pulsante HIGH ADDRESS venga rilasciato e, quindi, visualizza i valori degli interruttori. Questa esecuzione controlla se la risposta è giusta per il pulsante HIGH ADDRESS.

Quindi espanderemo il modulo relativo al pulsante MID ADDRESS in:

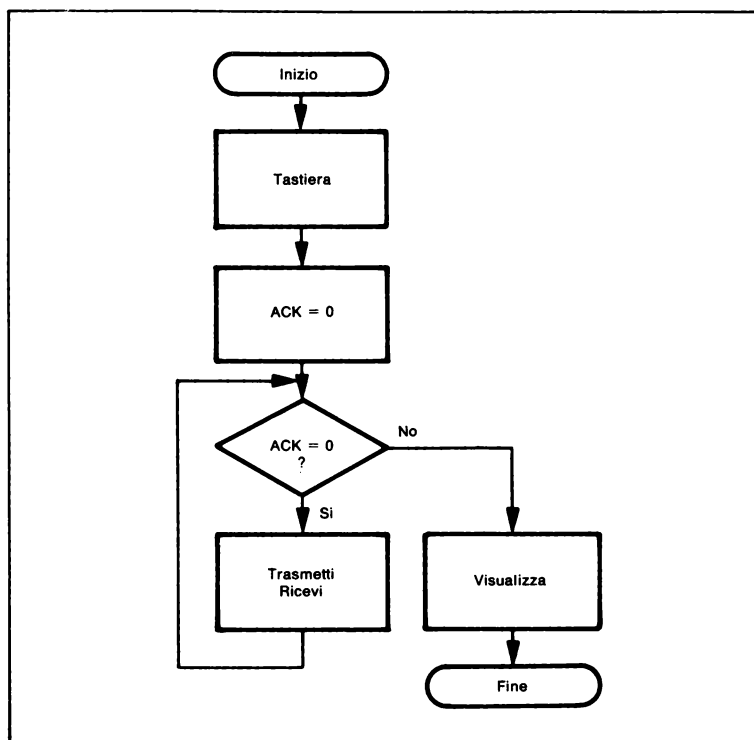
```
READ CPORT
MIDADDRBUTTON = (CPORT) AND MAMASK
```

Quando il pulsante MID ADDRESS viene rilasciato, il programma visualizzerà il valore degli interruttori. Questa esecuzione controlla se la risposta è corretta in rapporto al pulsante MID ADDRESS.

Quindi espanderemo il modulo del pulsante LOW ADDRESS in:

```
READ CPORT
LOADDRBUTTON = (CPORT) AND MASK
```

Quando il pulsante LOW ADDRESS viene rilasciato il programma visualizza i valori degli interruttori. Questa esecuzione controlla se la risposta è corretta per il pulsante LOW ADDRESS.

Figura 17-13. Diagramma di Flusso Iniziale del Terminale di Transazione.

Analogamente, espanderemo il modulo del pulsante DATA e controlleremo se la risposta è giusta. In questo modo avremo già collaudato l'intero programma.

Una volta espansi tutti i "monconi", avremo completato anche le fasi di codifica, debugging e collaudo. Naturalmente, dovremo stabilire con esattezza i risultati prodotti da ogni "moncone". Comunque, ad ogni livello individueremo molti nuovi errori logici, senza la necessità di ulteriori espansioni.

Progettazione Top-Down del Terminale di Verifica

Questo esempio, naturalmente, richiederà un numero maggiore di dettagli. **Cominceremo con il programma seguente** (cfr. Figura 17-13 per il diagramma di flusso):

```
KEYBOARD
ACK = 0
DO WHILE ACK = 0
        TRANSMIT
        RECEIVE
END
DISPLAY
```

In questo caso KEYBOARD, TRANSMIT, RECEIVE e DISPALY sono "monconi" di programmi che dovremo espandere. KEYBOARD, ad esempio, potrebbe mettere un numero a dieci cifre in un opportuno buffer, dopo averlo verificato.
La successiva fase di espansione produrrà il programma seguente per KEYBOAR RD (cfr. Figura 17-14):

```
VER = 0
DO WHILE VER = 0
      COMPLETE = 0
      DO WHILE COMPLETE = 0
       KEYIN
       KEYDS
END
VERIFY
END
```

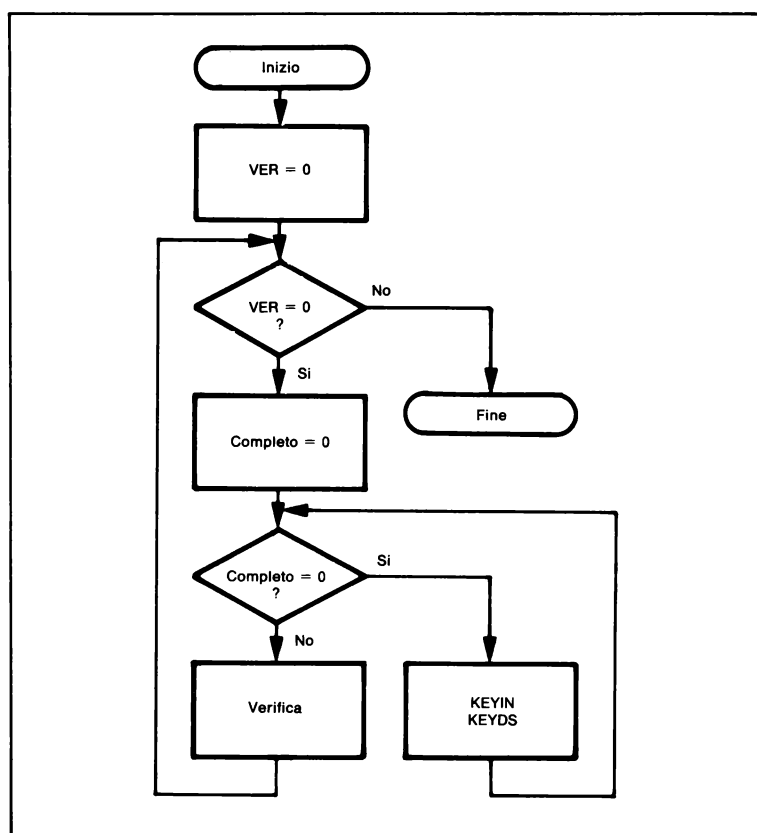



Figura 17-14. Diagramma di Flusso per l'espansione della Routine KEY-BOARD.

VER = 0 significa che non è avvenuto alcun inserimento. COMPLETE = 0 significa che l'inserimento è incompleto. KEYIN e KEYDS sono, rispettivamente, le routine di input e di visualizzazione. VERIFY controlla l'inserimento. Un "moncone" per KEYIN dovrebbe semplicemente mettere un inserimento casuale (prelevato da una tabella o da un generatore di numeri casuali) nel buffer e porre COMPLETE uguale a uno.

Definizione dei livelli

Continueremo applicando il solito procedimento di espansione, debugging e collaudo a TRANSMIT, RECEIVE e DISPLAY. È necessario espandere ciascun programma di un solo livello, in modo da non effettuare ogni volta l'integrazione di un programma intero. Dovete affidarvi al vostro buon senso per definire i livelli. Passi troppo piccoli sono una perdita di tempo, mentre passi troppo grandi riproporranno i problemi d'integrazione che la progettazione top-down avrebbe dovuto risolvere.

CONCLUSIONI

Fasi della progettazione Top-Down

La progettazione top-down porta un certo ordine nelle fasi di collaudo ed integrazione necessarie alla stesura di un programma. Fornisce un metodo sistematico per l'espansione di un diagramma di flusso o la definizione di un problema fino al livello necessario per la stesura definitiva del programma. Insieme con la programmazione strutturata, costituisce un insieme completo di tecniche di progettazione.

Al pari della programmazione strutturata, la progettazione top-down non è semplice. È necessario definire il problema con estrema chiarezza e lavorare passando sistematicamente da un livello all'altro. Anche in questo caso, la procedura risulterà noiosa, ma possiamo trarne un notevole vantaggio, soprattutto se ci atteniamo alle regole.
Ecco le varie fasi:

1. Iniziare con un diagramma di flusso molto generale
2. I "monconi" dovranno essere il più separati e completi possibile.
3. Definire con precisione tutti i risultati ottenibili da ogni "moncone" e sceglierne un gruppo da usare per il collaudo.
4. Controllare sistematicamente ogni livello.
5. Utilizzare le strutture della programmazionezione strutturata.
6. Espandere ogni "moncone" di un solo livello. Non cercare di fare troppe cose in un solo passaggio.
7. Controllare accuratamente le funzioni più utilizzate e le strutture dati.
8. Collaudare e verificare il tutto dopo l'espansione di ogni singolo "moncone". Non aspettare di aver completato un intero livello.
9. Conoscere con esattezza le possibilità dell'hardware. Non esitare a fermarsi ed a servirsi della progettazione bottom-up, ogni volta che appaia necessario.

PROGETTAZIONE DELLE STRUTTURE DATI

I programmatori alle prime armi raramente si preoccupano delle strutture dati, perchè pensano che i dati verranno messi da qualche parte nella memoria del computer, un pò come dei dischi messi uno sopra l'altro in un armadietto o dei libri su uno scaffale. Progettare delle strutture dati sarebbe come fare un'archiviazione completa dei propri dischi e dei propri libri: sono in pochi ad arrivare a tanto.

Ma resta il fatto che **molti sistemi computerizzati richiedono l'elaborazione di una enorme quantità di dati.** Gli algoritmi numerici presuppongono che il processore riesca facilmente a trovare un elemento della successiva riga o colonna di una matrice. I programmi di editing presuppongono che il processore possa trovare facilmente il carattere successivo, la riga precedente, una particolare stringa di caratteri o il punto d'inizio di una pagina o di un paragrafo. Un interfaccia utente per uno strumento di collaudo presuppone che il processore possa facilmente riconoscere un particolare comando o un dato introdotto e spostarlo da un posto all'altro. **Immaginate quanto sarebbe difficile realizzare le funzioni seguenti se i dati fossero semplicemente sparsi qua e là nella memoria o organizzati in un unico lungo vettore:**

1. L'operatore di una macchina utensile vuole inserire due ulteriori fasi di taglio, fra le fasi 14 e 15 di una sequenza complessiva di 40.
2. L'operatore di un impianto chimico vuole controllare gli ultimi dieci rilevamenti della temperatura all'imbocco del contenitore 05.
3. Un contabile vuole inserire un nuovo conto in un elenco ordinato alfabeticamente.

Il processore spesso passa la maggior parte del tempo a cercare i dati, passando dall'uno all'altro, e ad organizzarli.

SELEZIONARE LE STRUTTURE DATI

Criteri di scelta delle strutture di dati

Evidentemente, non possiamo fornire qui una descrizione completa di tutte le possibili strutture dati. Tenete presente che **nel caso di dati molto complessi, la progettazione delle strutture dati ha una notevole influenza sulla progettazione complessiva dei programmi. Riassumiamo brevemente alcune considerazioni che possono servire nella scelta della struttura più adatta:**

1. **Come sono correlati fra loro i vari dati?** Nel caso di elementi strettamente correlati, da un elemento si dovrebbe poter accedere a quello successivo, in quanto questo tipo di accesso sarà molto frequente.

2. **Quale tipo di operazioni verrà eseguito sui dati?** Semplici strutture lineari si riveleranno adeguate se i dati sono gestiti in un unico ordine prestabilito. Tuttavia, saranno necessarie strutture più complesse in caso operazioni come la ricerca, l'editing e l'ordinamento.
3. **Possono essere usate delle strutture standard?** Sono disponibili dei metodi standard per la gestione di strutture, come le code, gli stack e le liste collegate. In altri casi sarà necessario realizzare un programma specifico.
4. **Quale tipo di accesso è necessario?** È chiaro che avremo bisogno di una maggiore strutturazione se dobbiamo trovare elementi identificati da un numero o da una posizione relativa, invece che i primi o gli ultimi elementi di una lista. I dati devono essere organizzati in modo che l'accesso sia il più rapido possibile.

SOMMARIO

Come avrete notato, negli ultimi due capitoli non abbiamo parlato nè di un particolare microprocessore, nè del linguaggio assembly e non abbiamo scritto neppure una riga di codice. Tuttavia, adesso riguardo agli esempi ne sapete molto di più di quanto sarebbe accaduto se vi avessimo chiesto subito di scrivere i programmi. **Sebbene molti pensino che scrivere le istruzioni sia la parte fondamentale dello sviluppo del software, in realtà si tratta di una delle operazioni più facili.**

Una volta che avrete scritto alcuni programmi, la codifica diventerà estremamente semplice. Imparerete presto il set d'istruzioni, riconoscerete quali sono le istruzioni veramente utili e ricorderete le sequenze più comuni, che rappresentano la parte fondamentale di quasi tutti i programmi. **Vi accorgerete, allora, che sono altre le fasi difficili nello sviluppo del software ed esse non hanno neppure delle regole precise.**

Vi abbiamo suggerito alcuni metodi per procedere in modo sistematico durante le fasi iniziali. Al momento della definizione del problema è necessario stabilire tutte le caratteristiche del sistema: input, output, elaborazione, limitazioni di tempo e di memoria e gestione degli errori. **Bisogna considerare attentamente il modo in cui un sistema dovrà interagire con un altro sistema più grosso, del quale rappresenta una parte, e accertarsi se quel sistema prevede componenti elettriche, attrezzature meccaniche o un operatore umano. È fin da questa fase che bisogna rendere facile sia l'utilizzazione che la manutenzione.**

Nella fase di progettazione sono disponibili parecchie tecniche in grado di aiutarvi a specificare e documentare la logica del programma in modo sistematico. La programmazione modulare costringe a suddividere il programma in vari moduli separati, di dimensioni più piccole. La programmazione strutturata permette di definire la logica di questi moduli, mentre la progettazione top-down è un metodo sistematico per la loro integrazione ed il relativo collaudo. Naturalmente, nessuno vi

può costringere a seguire tutte queste tecniche; si tratta più che altro di indicazioni di carattere generale. Ma esse rappresentano uno standard di progettazione e dovreste considerarle come una base di partenza su cui sviluppare un vostro metodo personale.

BIBLIOGRAFIA

1. D.L. Parnas (vedi sotto) è stato uno dei maggiori esperti nell'ambito della programmazione modulare.
2. Raccolti da B. W. Unger (vedi sotto).
3. Formulati da D.L. Parnas.
4. K.J. Thurner and P.C. Patton. *Data Structures and Computer Architecture,* Lexington Books, Lexington, Mass., 1977.
5. K.S. Shankar. "Data Structures, Types, and Abstractions," *Computer*, Aprile 1980, pp. 67-77.

I testi seguenti forniscono ulteriori informazioni sulla definizione di un problema e la progettazione di un proramma:

Chapin, N. *Flowcharts*, Auerbach, princeton, N.J., 1971.
Dalton, W.F. "Design Microcomputer Software like Other Systems - Systematically," *Electronics*, Gennaio 1978, pp. 97-101.
Dijkstra, E.W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
Halstead, M.H. *Elements of Software Science,* American Elsevier, New York, 1977.
Hughes, J.K. and J.I. Michtom. *A Structured Approach to Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1977.
Morgan, D.E. and D.J. Taylor. "A Survey of Methods for Achieving Reliable Software," *Computer*, Febbraio 1977, pp. 44-52.
Myers, W. "The Need for Software Engineering," *Computer*, Febbraio 1978, pp. 12-25.
Parnas, D.L. "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM,* Dicembre 1972, pp. 1053-58.
Parnas, D.L. "A Technique for the Specification of Software Modules with Examples," *Communications of the ACM,* May 1973, pp. 330-336.
Phister, M. Jr. *Data Processing Technology and Economics,* Santa Monica Publishing Co., Santa Monica, Ca., 1976.
Schneider, V. "Prediction of Software Effort and Project Duration - Four New Formulas," *SIGPLAN Notice*, giugno 1978, pp. 49-59.

Schneiderman, B. et al. “Experimental Investigations of the Utility of Detailed Flowcharts in Programming”, *Communications of the ACM,* Giugno 1977, pp. 373-381.
Tausworthe, R.C. *Standardized Development of Computer software*. Prentice-Hall: Englewood Cliffs, N.J., 1977 (Part 1); 1979 (Part 2).
Unger, B.W. “Programming Languages for Computer System Simulation,” *Simulation*, Aprile 1978, pp. 101-10.
Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall: Englewood Cliffs, N.J., 1976.
Wirth, N. *Systematic Programming: an Introduction*. Prentice-Hall: Englewood Cliffs, N.J., 1973.
Yourdon, E.U. *Techniques of Program Structure and Design*. Prentice-Hall: Englewood Cliffs, N.J., 1975.

CAPITOLO 18

DOCUMENTAZIONE

Sviluppare software non significa soltanto realizzare un programma funzionante, ma anche provvedere ad una adeguata documentazione, che ne consenta l'uso, la manutenzione e l'ampliamento. Una buona documentazione facilita anche la verifica ed il collaudo di un programma ed è essenziale nelle successive fasi del suo ciclo vitale.

PROGRAMMI CHE SI AUTO-DOCUMENTANO

Sebbene nessun programma si possa documentare da solo in modo esauriente, alcune regole, che abbiamo già menzionato, possono rivelarsi utili. Ricordiamole:

- Una struttura chiara e semplice con il minor numero possibile di trasferimenti di controllo (salti).
- Uso di nomi e label che abbiano un significato
- Nomi, anzichè numeri, per i dispositivi di I/O, i parametri, i fattori numerici, gli indirizzi delle subroutine, ecc.
- Cercare di ottenere il massimo della semplicità, anche se questo richiede della memoria in più e comporta un leggero aumento del tempo di esecuzione.

Ad esempio, il programma seguente invia un carattere ad una telescrivente:

```
          MOVEQ     -1,D0
          MOVE.B    $6000,D0
          MOVEQ     #10,D2

          BCLR.B    #7,0(A0)
SNDBIT    BSR       DELAY9_1
          ROR.W     #1,D0
          BCS.S     SNDONE
          BCLR.B    #7,0(A0)
          BRA.S     NEXT

SNDONE    BSET.B    #7,0(A0)
NEXT      DBRA      D2,SNDBIT

          RTS
          END
```

Anche senza commenti possiamo migliorare il programma nel modo seguente:

```
PROGRAM  EQU      $4000
DATA     EQU      $6000

PIADA    EQU      $00                   OFFSET PER IL REGI-
                                        STRO DATI A

TTYBIT   EQU      $07                   TTY COLLEGATA AL
                                        BIT 7
CHRBIT   EQU      $08                   NUMERO DI BIT PER
                                        CARATTERE
STPBIT   EQU      $02                   NUM. BIT STOP DA
                                        TRASMETTERE

TTYOUT   MOVEQ    -1,D0                 FORMA BIT DI STOP
         MOVE.B   CHAR,D0               PRENDI L'OUTPUT
                                        DELLA TTY
         MOVEQ    #1+CHRBIT             CONT.BIT CORRETTO
                  +STPBIT-1,D2          PER DBRA

         BCLR.B   #TTYBIT,PIADA(A0)     INVIA IL BIT DI START
SNDBIT   BSR      DELAY9_1              ATTENDI IL TEMPO DI
                                        1 BIT
         ROR.W    #1,D0                 CARRY=BIT SUC-
                                        CESSIVO
         BCS.S    SNDONE                SE DATA=1 ALLORA
                                        INVIA UN '1'

         BCLR.B   #TTYBIT,PIADA(A0)     INVIA '0' COME BIT
                                        DEL DATO
         BRA.S    NEXT

SNDONE   BSET.B   #TTYBIT,PIADA(A0)     INVIA '1' COME BIT
                                        DEL DATO
NEXT     DBRA     D2,SNDBIT             CONTINUA FINCHÈ
                                        NON SONO
                                        STATI INVIATI TUTTI I
                                        BIT
         RTS
         END
```

Questa versione è indubbiamente più comprensibile di quella precedente. **Anche senza un'ulteriore documentazione, è possibile indovinare la funzione del programma ed il significato della maggior parte delle variabili.** Un certo grado di auto-documentazione è indispensabile.

Ecco alcune indicazioni sulla scelta dei nomi:

1. **Usare un nome chiaro** quando è disponibile, come TTY o CRT per i dispositivi di output, START o RESET per gli indirizzi, RITARDO o SORT per le subroutine, COUNT o LENGTH per i dati.
2. **Evitare le sigle,** come S16BA per SORT 16-BIT ARRAY. Per molti risulteranno prive di significato.
3. **Usare parole intere,** o quasi, quando è possibile, come FATTO, STAMPA, INVIO, ecc.
4. **Usare nomi più diversi possibile.** Evitare nomi che sembrano uguali, come TEMPI e TEMP1, o che assomigliano ai codici operativi o alle direttive dell'assemblatore.

COMMENTI

Come scrivere buoni commenti

I commenti sono un modo semplice, con il quale fornire un'ulteriore documentazione. Tuttavia, sono pochi i programmi (anche quelli usati come esempi nei manuali) che hanno dei commenti esaurienti. Se volete scrivere dei buoni commenti, cercate di seguire queste indicazioni:

1. Non spiegare gli effetti interni di un'istruzione, ma solo la funzione di quell'istruzione nell'ambito del programma. Commenti del tipo

```
SUBQ.W #1,D0  D0 := D0 - 1
```

non aiutano a capire il programma. Un commento più utile sarà

```
SUBQ.W #1,D0   NUMERO DI LINEA = NUMERO DI LINEA - 1
```

Non dimenticate che ci sono degli appositi manuali che spiegano in che modo il processore esegue le varie istruzioni. I commenti servono a spiegare quali funzioni il programma sta svolgendo e quali metodi impiega.

2. **I commenti devono essere molto chiari.** Non usare abbreviazioni o sigle, a meno che non siano ben note (come ASCII, PIA o UART) o comprensibili a tutti (come "num" per numero, "ms" per millisecondi, ecc.). Evitare commenti del tipo

```
SUBQ.W #1,D0  LN := LN - 1
```

o

```
SUBQ.W #1,D0  DEC, LN DI 1
```

Vale sempre la pena battere qualche riga di commento in più.

3. **Commentare ogni punto importante o oscuro.** In particolare è necessario indicare operazioni che non hanno un significato molto chiaro, come

```
MOVEA.L (A0),A0   PRENDI IND.ELEMENTO SUCCESS. DELLA
CODA
o
ANDI.B #$FE,PIADA(A0)   SPEGNI L'INDICATORE A LED
```

   Chiaramente, le operazioni di I/O richiedono spesso dei commenti estesi. Se non siete sicuri di che cosa faccia una certa istruzione o se dovete pensarci sopra, aggiungete un commento che la chiarisca e che vi risparmi del tempo in seguito, oltre ad esservi di aiuto per la documentazione.
4. **Non commentare le cose ovvie.** Un commento su ogni riga rende difficile individuare le cose importanti. Le istruzioni standard come

```
DBRA D1,LOOP
```

   non hanno nessun bisogno di essere commentate, a meno che non ne facciate un uso particolare. Spesso un commento è sufficiente per parecchie righe di programma, come in questo caso

```
CLR.B     PIACA(A0)                 INIZIALIZZA LA PARTE A
MOVE.B    #A_DATDIR,PIADDA(A0)
MOVE.B    #A_CNTRL,PIACA(A0)
o
MOVE.B    (A0)+,D0                  SCAMBIARE IL BYTE PIÙ SI-
                                    GNIF.
MOVE.B    (A0),-(A0)                ...E QUELLO MENO SIGNIFI-
                                    CATIVO
MOVE.B    D0,1(A0)
```

5. **Mettere i commenti sulle righe cui si riferiscono oppure all'inizio di una sequenza.**
6. **Aggiornare i commenti.** Quando modificate il programma cambiate anche i commenti.
7. **Utilizzare termini ed espressioni standard** per i commenti. Non preoccupatevi delle ripetizioni. Nomi diversi per indicare le stesse cose provocano confusione, anche se si tratta di variazioni minime, come fra COUNT e COUNTER, START e BEGIN, DISPLAY e LEDS o PANEL e SWITCHES. L'incoerenza non paga mai. Delle piccole variazioni vi sono chiare adesso, ma possono non esserlo inseguito; altri si confonderanno subito.
8. **I commenti mescolati alle istruzioni devono essere brevi.** Lasciate la spiegazione completa ai commenti iniziali o ad altra documentazione, altrimenti il programma scompare in mezzo ai commenti e diventa difficile individuare le istruzioni.

9. **Perfezionare i commenti.** Se c'è un commento che non riuscite a leggere o a capire, perdete un pò di tempo per cercare di cambiarlo. Se il listato sta diventando troppo fitto, saltate alcun righe. I commenti non migliorano da soli; in realtà, se li trascurate e dimenticate esattamente ciò che avete fatto, finiscono per peggiorare sempre di più.
10. **Servirsi dei commenti per mettere delle intestazioni all'inizio di ogni sezione, sottosezione o subroutine più importante.** Questo serve a indicare le funzioni del codice che segue, a dare informazioni riguardo all'algoritmo impiegato, agli input, agli output e ad eventuali effetti secondari che si possono produrre.
11. **Quando si modifica un programma che funziona, è opportuno usare dei commenti per descrivere le modifiche apportate, indicando data e autore della nuova versione.** Queste informazioni dovranno trovarsi all'inizio del programma (in modo che un qualsiasi utente possa facilmente distinguere una versione dall'altra) e nei punti che sono stati cambiati.

Non dimenticate che **i commenti sono importanti. Dei buoni commenti vi risparmieranno tempo e fatica.** Dedicate un pò del vostro tempo ai commenti e cercate di renderli più chiari possibile.

ESEMPI

18-1. Commentare la routine di output di una telescrivente

Il programma fondamentale è:

```
         MOVEQ    -1,D0
         MOVE.B   $6000,D0
         MOVEQ    #10,D2

         BCLR.B   #7,0(A0)
SNDBIT   BSR      DELAY9_1
         ROR.W    #1,D0
         BCS.S    SNDONE

         BCLR.B   #7,0(A0)
         BRA.S    NEXT

SNDONE   BSET.B   #7,0(A0)
NEXT     DBRA     D2,SNDBIT

         RTS
         END
```

Commentando i punti più importanti e sostituendo i numeri con i nomi, avremo:

```
* OUTPUT AD UNA TELESCRIVENTE
* QUESTO PROGRAMMA INVIA IL CARATTERE NELLA LOCAZIONE CHAR
* ALLA TELESCRIVENTE IL CUI INDIRIZZO È NEL REG. A0


PROGRAM  EQU     $4000
DATA     EQU     $6000

PIADA    EQU     $00                   OFFSET PER IL REGI-
                                       STRO DATI A DEL PIA

TTYBIT   EQU     $07                   TTY COLLEGATA AL BIT
                                       7
CHRBIT   EQU     $08                   NUMERO DI BIT PER CA-
                                       RATTERE
STPBIT   EQU     $02                   NUM. BIT STOP DA TRA-
                                       SMETTERE


         ORG     DATA

CHAR     DS.B    1                     CARATTERE DI OUTPUT
                                       ALLA TTY

         ORG     PROGRAM

TTYOUT   MOVEQ   -1,D0                 FORMA BIT DI STOP
         MOVE.B  CHAR,D0               PRENDI L'OUTPUT DEL-
                                       LA TTY
         MOVEQ   #1+CHRBIT+STPBIT-
                 1,D2                  CONT.BIT CORRETTO
                                       PER DBRA

         BCLR.B  #TTYBIT,PIADA(A0)     INVIA IL BIT DI START
SNDBIT   BSR     DELAY9__1             ATTENDI IL TEMPO DI 1
                                       BIT
         ROR.W   #1,D0                 CARRY=BIT SUCCESSI-
                                       VO
         BCS.S   SNDONE                SE DATA=1 ALLORA IN-
                                       VIA UN '1'

         BCLR.B  #TTYBIT,PIADA(A0)     INVIA '0' COME BIT DEL
                                       DATO
         BRA.S   NEXT

SNDONE   BSET.B  #TTYBIT,PIADA(A0)     INVIA '1' COME BIT DEL
                                       DATO
NEXT     DBRA    D2,SNDBIT             CONTINUA FINCHÈ NON
                                       SONO
                                       STATI INVIATI TUTTI I BIT

         RTS
         END
```

Modificare il Programma

Possiamo facilmente modificare questo programma, in modo da trasferire un'intera stringa di dati, che inizi alla locazione CHRSTR e finisca con un carattere 03 (ASCII ETX).

Programma 18-1 vers. 3

Dei buoni commenti vi aiuteranno a modificare il programma in modo da soddisfare nuove esigenze. Ad esempio, provate a cambiare il programma precedente in modo che:

- Ogni messaggio inizi con il carattere ASCII STX (02), seguito da un codice di identificazione a due cifre, memorizzato nella locazione IDCODE.
- Non siano aggiunti dei bit di Start o di Stop
- Fra un bit e l'altro vi sia un intervallo di 1 ms.
- Vengano trasmessi 40 caratteri, a partire da quello che si trova all'indirizzo in DPTR.
- Ogni messaggio termini con due caratteri ASCII ETX (03) consecutivi.

18-2. COMMENTARE UNA ROUTINE DI ADDIZIONE IN PRECISIONE MULTIPLA

4 Il programma di partenza è:

```
        ORG      $4000
        MOVE.L   #$6008,A0
        MOVE.L   #$6208,A1
        MOVE     #0,CCR
        MOVEQ    #7,D2
LOOP    MOVE.B   -(A0),D0
        MOVE.B   -(A1),D1
        ADDX.B   D1,D0
        MOVE.B   D0,(A0)
        DBRA     D2,LOOP
        RTS
        END
```

Punti Importanti

Per prima cosa, bisogna commentare i punti più importanti. Si tratta, in genere, di inizializzazioni, prelievi di dati ed operazioni di elaborazione. Non vi preoccupate delle sequenze standard come l'aggiornamento dei puntatori o dei contatori. Non dimenticate che **i nomi sono più chiari dei numeri, perciò usateli spesso.**

La nuova versione del programma è:

```
*           ADDIZIONE IN PRECISIONE MULTIPLA
*
*           QUESTO PROGRAMMA SOMMA DUE NUMERI POSTI
*           NELLE LOCAZIONI NUM1 E NUM2 E
*           SALVA IL RISULTATO NELLA LOCAZIONE NUM1
*
*           I NUMERI DEVONO ESSERE DI 8 BIT
*           (OPPURE CAMBIARE BYTECOUNT)

PROGRAM     EQU       $4000

NUM1        EQU       $6000
NUM2        EQU       $6200
BYTECOUNT   EQU       $8

            ORG       PROGRAM

            MOVEA.L   #NUM1+BYTE        IND. OLTRE FINE
                      COUNT,A0          PRIMO NUM.
            MOVEA.L   #NUM2+BYTE        IND. OLTRE FINE
                      COUNT,A1          SECONDO NUM.
            MOVE      #0,CCR
            MOVEQ     #BYTECOUNT-1,D2

LOOP        MOVE.B    -(A0),D0          PRENDI I BYTE DA SOM-
                                        MARE PRIMA
            MOVE.B    -(A1),D1          QUELLI MENO SIGNIFI-
                                        CATIVI
            ADDX.B    D1,D0             SOMMA CON CARRY
            MOVE.B    D0,(A0)           SALVA IL RISULTATO IN
                                        NUM1
            DBRA      D2,LOOP

            RTS
```

Funzioni poco Chiare

In secondo luogo, è opportuno controllare se ci sono delle istruzioni che non hanno un significato chiaro e spiegarne gli scopi con dei commenti. In questo caso lo scopo di MOVE #0,CCR è di azzerare il flag di Extend (e gli altri flag) prima di sommare i byte meno significativi.

```
*           ADDIZIONE IN PRECISIONE MULTIPLA
*
*           QUESTO PROGRAMMA SOMMA DUE NUMERI POSTI
*           NELLE LOCAZIONI NUM1 E NUM2 E
*           SALVA IL RISULTATO NELLA LOCAZIONE NUM1
*
*           I NUMERI DEVONO ESSERE DI 8 BIT
*           (OPPURE CAMBIARE BYTECOUNT)
```

```
PROGRAM     EQU       $4000

NUM1        EQU       $6000              IND. DEL PRIMO NUME-
                                         RO BINARIO
NUM2        EQU       $6200              IND. DEL SECONDO
                                         NUM. BINARIO
BYTECOUNT   EQU       $8                 NUMERO DEI BIT DA
                                         SOMMARE

            ORG       PROGRAM
            MOVEA.L   #NUM1+BYTE         IND. OLTRE FINE
                      COUNT,A0           PRIMO NUM.
            MOVEA.L   #NUM2+BYTE         IND. OLTRE FINE
                      COUNT,A1           SECONDO NUM.
            MOVE      #0,CCR             AZZERA FLAG EXTEND E
                                         GLI ALTRI FLAGS
            MOVEQ     #BYTECOUNT-1,D2    CONT. LOOP CORRETTO
                                         PER DBRA

LOOP        MOVE.B    -(A0),D0           PRENDI I BYTE DA SOM-
                                         MARE. PRIMA
            MOVE.B    -(A1),D1           QUELLI MENO SIGNIFI-
                                         CATIVI.
            ADDX.B    D1,D0              SOMMA CON CARRY
            MOVE.B    D0,(A0)            SALVA IL RISULTATO IN
                                         NUM1
            DBRA      D2,LOOP

            RTS
```

Domande per i Commenti

In terzo luogo, bisogna chiedersi se i commenti dicono ciò che è necessario sapere per usare il programma; ad esempio:

1. Qual è il punto d'ingresso del programma? Ne esistono di alternativi?
2. Quali parametri sono necessari? Come ed in quale forma devono essere forniti?
3. Quali operazioni effettua il programma ?
4. Da dove vengono prelevati i dati?
5. Dove vengono messi i risultati?
6. Quali sono i casi speciali previsti?
7. Cosa fa il programma quando si verifica un errore?
8. Qual è il punto di uscita?

Alcune domande possono apparire inutili ed alcune delle risposte già scontate. Assicuratevi, però, **di non dover sezionare il programma per rispondere a delle domande importanti, senza dimenticare, tuttavia, che un numero eccessivo di spiegazioni può essere un ostacolo all'utilizzazione del programma.** Ci sono delle modifiche che vorreste apportare al listato? Non esitate a farle: tocca a voi stabilire se il commento è adeguato e sufficiente.

DIAGRAMMI DI FLUSSO COME DOCUMENTAZIONE

Abbiamo già descritto l'impiego dei diagrammi di flusso come strumento di progettazione nel Capitolo 17. I diagrammi di flusso si rivelano utili anche in fase di documentazione, soprattutto se:

- Non sono confusi o troppo dettagliati.
- I punti in cui è richiesta una decisione sono spiegati ed indicati con chiarezza.
- Comprendono tutte le diramazioni.
- Corrispondono ai listati reali dei programmi.

I diagrammi di flusso sono utili quando forniscono un quadro complessivo del programma. Se risultano difficili da leggere al pari di un listato, non servono a niente.

PROGRAMMI STRUTTURATI COME DOCUMENTAZIONE

Un programma strutturato può servire a commentare un programma in linguaggio assembly se:

- Nei commenti sono indicati gli scopi di ogni sezione.
- Appare chiaro quali statement fanno parte di ogni struttura condizionale o loop, attraverso l'indentazione e gli indicatori di fine struttura.
- La struttura complessiva è stata semplificata al massimo.
- Viene usato un linguaggio coerente e ben definito.

Portabilità dei programmi strutturati

Il programma strutturato può servire a controllare la logica ed eventualmente a migliorarla. Inoltre, dal momento che è indipendente dalla macchina, serve anche a realizzare una stessa funzione su un altro computer.

MAPPE DI MEMORIA

Scopo delle mappe di memoria

Una mappa di memoria è semplicemente un elenco di tutte le assegnazioni di memoria presenti in un programma. Consente di stabilire la memoria necessaria, le locazioni riservate ai dati o alle subroutine e le parti della memoria che non sono state assegnate. Serve per individuare rapidamente le locazioni destinate a memorizzazioni temporanee e i punti di ingresso e, inoltre, consente l'uso della memoria a differenti routine o, additittura, a programmatori diversi. Una mappa di memoria consentirà, anche, un facile accesso ai dati ed alle subroutine, nel caso di eventuali ampliamenti o in fase di manutenzione. **Qualche volta uno schema grafico si rivela più utile di un semplice elenco.**

Questa è una mappa tipica:

Memoria di Programma

Indirizzo	Routine	Scopo
E000 - E1FF	RDKBD	Servizio Interrupt Tastiera
E200 - E240	BRKPT	Breakpoint via Software
E241 - E250	DELAY	Ritardo Generalizzato
E251 - E270	DSPLY	Controllo Display Operatore
E271 - E3EF	SUPER	Programma Principale Superiore
0000 - 03FF		Vettori Reset e Interrupt

Memoria Dati

Indirizzo	Nome	Scopo
1000	NKEYS	Numero dei Tasti Premuti
1001 - 1002	KBPTR	Puntatore Buffer tastiera
1003 - 1041	KBUFFR	Buffer Tastiera
1042 - 1050	DBUFFR	Buffer Display
1051 - 106F	TEMP	Memorizzazione Temporanea
1070 - 10FF	STACK	Stack Hardware

La mappa può elencare anche punti d'ingresso addizionali ed includere una descrizione specifica delle parti di memoria non utilizzate.

LISTE DI PARAMETRI E DEFINIZIONI

Liste di parametri e definizioni all'inizio del programma principale e di ciascuna subroutine rendono il listato più comprensibile e più semplici le eventuali modifiche. Queste sono le regole da seguire:

1. **Indicare separatamente le locazioni dei dati, le unità di I/O, i parametri, le definizioni e gli indirizzi costanti.**

2. **Disporre gli elenchi in ordine alfabetico, se possibile, aggiungendo una descrizione per ogni voce.**
3. **Dare un nome a tutti i parametri che potrebbero cambiare ed includerli nell'elenco.** Ci riferiamo alle costanti di tempo, ad input o codici corrispondenti a dei tasti o a delle funzioni particolari, a sequenze di bit utilizzate come maschere o come controllo, a caratteri d'inizio o di fine, a valori soglia, ecc.
4. **Elencare separatamente gli indirizzi di memoria costanti,** come gli indirizzi delle routine di reset o di quelle destinate a servire gli interrupt, l'indirizzo iniziale del programma e dell'area di memoria utilizzata come stack, ecc.
5. **Assegnare un nome ad ogni porta utilizzata da un dispositivo di I/O,** anche se una stessa porta è comune a diversi dispositivi. La distinzione renderà più semplice espandere o modificare la sezione di I/O.

Un tipico elenco di definizioni è il seguente:

```
*
* COSTANTI DI MEMORIA DEL SISTEMA
*
IRQ__1LEV   EQU   $21000   ROUTINE DI SERVIZIO INTERRUPT
                           LIV. 1
IRQ__2LEV   EQU   $210A8   ROUTINE DI SERVIZIO INTERRUPT
                           LIV. 2
IRQ__7LEV   EQU   $22000   ROUTINE DI SERVIZIO INTERRUPT
                           LIV. 7
MEMORIA     EQU   $0       IND. INIZIALE DELLA MEMORIA
SSTKPNT     EQU   $F000    PUNTATORE INIZIALE STACK SU-
                           PERVISORE
USTKPNT     EQU   $E000    PUNTATORE INIZIALE STACK U-
                           TENTE
*
*  UNITÀ DI I/O
*
PIA1        EQU   $3FF40   INDIRIZZO DI BASE DEL PIA 1
PIA2        EQU   $3FF41   INDIRIZZO DI BASE DEL PIA 2
ACIA1       EQU   $3FF01   INDIRIZZO DI BASE DELL'ACIA 1
ACIA2       EQU   $3FF21   INDIRIZZO DI BASE DELL'ACIA 2
*
*  OFFSET DELLE UNITÀ DI I/O
*
PIADDA      EQU   $0       OFFSET PER IL REG. DIREZIONE
                           DATI A
PIADA       EQU   $0       OFFSET PER IL REG. DATI A
PIACA       EQU   $4       OFFSET PER IL REG. DI CONTROL-
                           LO A
*
```

```
*  SPAZIO PER LA MEMORIZZAZIONE DI DATI
*
            ORG      RAM
NUMROWS     DS.B     1          NUM. DI RIGHE DELLA TASTIERA
                                DI INPUT
NUMCOL      DS.B     1          NUM. DI COLONNE TASTIERA DI IN-
                                PUT
INPUTBUF    DS.L     1          INDIRIZZO DEL BUFFER DI INPUT
OUTBUF      DS.L     1          INDIRIZZO DEL BUFFER DI OUTPUT
TEMP        DS.L     $10        BUFFER DATI TEMPORANEI
*
*  PARAMETRI
*
BOUNCE1     EQU      $2         TEMPO RIMBALZO TASTIERA IN
                                MS
OPEN        EQU      $0F        SEQ. INPUT SE NESSUN TASTO È
                                PREMUTO
DISDLY      EQU      $01        LUNGH. IMPULSO PER I DISPLAY IN
                                MS.
*
*  DEFINIZIONI
*
ALLHI       EQU      $FF        INPUT FORMATO DA TUTTI UNO
STCON       EQU      $80        OUTPUT PER INIZIO IMPULSO
                                CONVERSIONE
```

Naturalmente, **le definizioni delle locazioni utilizzate per memorizzare dati non saranno sempre in ordine alfabetico, in quanto il progettista può modificarne l'ordine per vari motivi.**

ROUTINE D'ARCHIVIO

Una documentazione standard delle subroutine permetterà di formare una libreria di programmi immediatamente disponibili. Descrivendo ciascuna subroutine in una forma standard, chiunque può rendersi conto, con una semplice occhiata, di quale funzione svolge una certa routine e in che modo poterla usare. Un archivio di questo tipo deve essere organizzato molto accuratamente, definendo ciascuna routine in base al tipo di processore, di linguaggio e di programma. Senza un'organizzazione ed una documentazione adeguate, l'uso di una libreria di questo tipo potrà risultare più difficile che scrivere un programma ex novo. Se intendete utilizzare delle subroutine tratte da un archivio o scritte da altri, ne dovrete conoscere tutti i possibili effetti, in modo da poter, poi, effettuare il debugging dell'intero programma.

COMMENTARE LE ROUTINE D'ARCHIVIO

Queste sono alcune delle informazioni da indicare per ciascuna routine:

- Scopo del programma
- Processore usato
- Linguaggio usato
- Paramentri necessari e modalità per il loro passaggio alla subroutine
- Risultati prodotti e relative modalità di passaggio al programma principale
- Numero di byte utilizzati
- Numero di cicli di clock richiesti. Può trattarsi di un valore medio o tipico e, in certi casi, varia notevolmente. L'effettivo tempo di esecuzione dipenderà, naturalmente, dalla frequenza di clock del processore e dalla velocità di accesso alla memoria.
- I registri interessati
- I flag eventualmente modificati
- Un esempio tipico
- La gestione degli errori
- I casi speciali
- Il listato documentato del programma

Documentazione di programmi complessi

Se il programma è particolarmente complesso bisogna includere anche un diagramma di flusso generale o una descrizione in forma strutturata. Come abbiamo già ricordato, le routine di questo tipo si rivelano molto più utili quando svolgono una singola funzione in modo generico e, quindi, sono utilizzabili in varie situazioni.

DOCUMENTAZIONE COMPLESSIVA

Una documentazione completa includerà tutti o quasi gli elementi che abbiamo menzionato.

PACKAGE DI DOCUMENTAZIONE

Un package di documentazione completo comprende:

- Diagrammi di flusso generali
- Una descrizione del programma
- Una lista di tutti i parametri e di tutte le definizioni
- Una mappa di memoria
- Un listato documentato del programma
- Una descrizione delle modalità di collaudo ed i relativi risultati

Della documentazione possono far parte anche:

- Diagrammi di flusso del programma
- Diagrammi di flusso dei dati
- Programmi strutturati

Anche questo package è sufficiente solo **per il software** che non è **destinato alla produzione.** Altrimenti **sono necessari anche i seguenti manuali:**

- Manuale della Logica del Programma
- Guida Utente
- Manuale di manutenzione

Manuale della Logica del Programma

Il manuale relativo alla logica del programma amplia le spiegazioni scritte, fornite con il software. Indica gli scopi che si è voluto perseguire nella progettazione del sistema, gli algoritmi utilizzati e quali aspetti si è voluto privilegiare, presupponendo che il lettore disponga di una certa competenza tecnica, ma non abbia una conoscenza dettagliata del programma. Dovrebbe fornire una guida alle varie funzioni del programma, oltre a spiegare la struttura dei dati e la loro manipolazione.

Guida Utente

Funzione della guida utente

La Guida Utente è l'elemento più importante della documentazione. Anche se il sistema è stato ben progettato, non sarà abbastanza utile se non si riesce a capirne le funzioni ed a sfruttarne le caratteristiche. **La Guida Utente dovrà spiegare le caratteristiche del sistema ed il loro uso, fornire parecchi esempi che chiariscano il testo e dare indicazioni sui vari procedimenti da seguire. Scrivere una Guida Utente richiede chiarezza ed obiettività, dal momento che bisogna mettersi nei panni di un profano.**

Bisogna evitare di scoraggiare un principiante o mettere alla prova la pazienza di un utente esperto: due versioni distinte potrebbero risolvere questo problema. **Una guida per il principiante illustrerà le caratteristiche più comuni del programma con l'aiuto di semplici esempi e descrizioni dettagliate. Una guida destinata all'utente esperto fornirà delle descrizioni più ampie delle caratteristiche del sistema ed un numero minore di esempi.** Un principiante ha bisogno di essere aiutato anche ad avviare il sistema, mentre un esperto vuole solo un manuale che gli serva come riferimento occasionale.

Manuale di Manutenzione

Il manuale di manutenzione è progettato per il programmatore che deve modificare il sistema. **Contiene le procedure necessarie per qualsiasi modifica o ampliamento, definite durante la fase di progettazione.**

IMPORTANZA DELLA DOCUMENTAZIONE

La documentazione non è un aspetto secondario e non la dobbiamo rimandare all'ultimo momento. Una documentazione esauriente, insieme a buone abitudini di programmazione, non è soltanto un elemento importante del prodotto finito, ma rende anche più semplice, rapido e produtttivo il processo di sviluppo. **Un progettista deve essere consapevole che una documentazione completa e coerente dovrà accompagnare tutte le fasi di sviluppo del software.**

CAPITOLO 19

DEBUGGING

Come abbiamo osservato all'inizio di questa sezione, il debugging (verifica) ed il collaudo sono fra le fasi più lunghe dello sviluppo del software. **Sebbene metodi come quelli della programmazione modulare e strutturata o della progettazione top-down abbiano semplificato molto i programmi e ridotto la percentuale di errori, il debugging ed il collaudo restano ancora molto complessi,** anche perchè definiti in modo molto approssimativo. La scelta di un insieme di dati adatto a collaudare un sistema è, raramente, il risultato di una vera analisi scientifica. Talvolta, trovare gli errori è come giocare ad attaccare la coda dell'asino, solo che, in questo caso, l'asino si muove ed un programmatore deve servirsi di un telecomando per mettere la coda al posto giusto. Spesso il debugging è un'esperienza molto frustrante.

In questo capitolo verrano descritti, innanzitutto, gli strumenti da utilizzare in fase di debugging. Quindi, passeremo in rassegna le procedure fondamentali da seguire ed i più comuni tipi di errore, fornendo alcuni esempi concreti. Nel capitolo successivo parleremo della scelta di opportuni dati di controllo e del collaudo dei programmi.

Di gran parte dei metodi di debugging descriveremo solamente le finalità. Non esistono degli standard in questo settore e, del resto, non possiamo prendere in esame tutti i prodotti disponibili. Gli esempi mostreranno l'impiego, i vantaggi e le limitazioni di alcuni degli strumenti più diffusi.

Funzioni degli strumenti di debugging

Gli strumenti di debugging hanno due funzioni principali. Una è quella di individuare la sezione del programma che contiene l'errore; l'altra consiste nel fornire indicazioni più dettagliate, rispetto ad una normale esecuzione, riguardo a ciò che il computer sta facendo, in modo da evidenziare la causa dell'errore. Gli attuali strumenti di debugging non trovano e gli correggono errori da soli: è necessario conoscere il funzionamento di un programma per individuare un errore e poterlo, poi, correggere.

Fra quelli utilizzati più spesso ricordiamo:

- Il metodo del breakpoint (punto di arresto)
- L'esecuzione single-step (un'istruzione per volta)
- La funzione Trace
- Un programma che visualizza il contenuto dei registri
- Un programma che visualizza il contenuto della memoria

BREAKPOINT

Un breakpoint (o punto di arresto) è un punto in cui il programma si ferma automaticamente in modo che l'utente possa esaminare lo stato del sistema. Il programma non riprenderà l'esecuzione finchè l'utente non darà il relativo comando. I breakpoint permettono di controllare o aggirare intere sezioni di un programma. Per verificare se una routine di inizializzazione è corretta, vi metteremo un breakpoint alla fine ed eseguiremo il programma. In questo modo, diventa possibile controllare le locazioni di memoria ed i registri, per vedere se l'intera sezione è corretta. Se non lo è, bisognerà localizzare l'errore con dei punti di arresto più a monte o mediante l'esecuzione single-step.

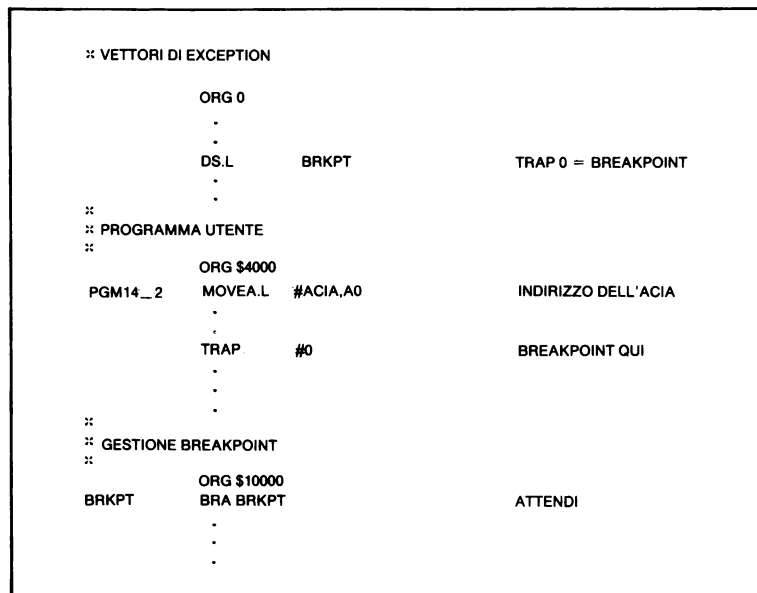
Uso delle exceptions per realizzare breakpoints

I breakpoint utilizzano spesso il sistema di processo delle Exception (cfr. Capitolo 15). Per i breakpoint possiamo servirci di uno qualsiasi dei 16 vettori di Trap oppure di uno dei 7 livelli di interrupt. In quest'ultimo caso, è necessario un dispositivo esterno che generi il relativo segnale. Di solito, un breakpoint provoca l'esecuzione di uno speciale programma, che, ad esempio, stampa automaticamente i contenuti di determinati registri oppure attende che l'utente dia un comando.

Inserimento dei Breakpoint

Il modo più semplice per inserire un breakpoint in un programma è quello di sostituire la prima word di un'istruzione con un'istruzione Trap. Quando questa viene eseguita, il controllo del programma passa alla routine di breakpoint indicata dal relativo vettore di Trap, viene selezionato il modo Supervisore e vengono salvati il contenuto del contatore di programma e quello del registro di stato.

Non bisogna dimenticare che il valore del contatore di programma, che è stato salvato, indica l'istruzione successiva a quella che ha provocato la Trap. Se desideriamo che sia visualizzato l'indirizzo effettivo del breakpoint o che il programma riprenda correttamente, dopo aver ripristinato l'istruzione originale, dovremo sottrarre due dal valore del contatore di programma. Il modo più semplice è

Figura 19-1. Una Semplice Routine di Breakpoint.

quello di impiegare l'istruzione SUBQ.L #2,-2(A7). Questo metodo presuppone che il puntatore dello stack Supervisore indichi ancora i dati salvati al momento dell'istruzione Trap.

La Figura 19-1 mostra una semplice routine di breakpoint con il suo vettore di Trap e la relativa chiamata. Questa routine provoca un loop senza fine ed il solo modo per uscirne è quello di servirsi di un reset o di un interrupt.

Metodi per Inserire e rimuovere Breakpoint

Breackpoints in ROM e in PROM

Molti monitor danno la possibilità di inserire e rimuovere automaticamente dei breakpoint, mediante delle istruzioni Trap. Questi breakpoint non influiscono sul tempo di esecuzione del programma, finchè uno di essi non viene eseguito; tuttavia, non è possibile sostituire delle istruzioni che si trovano in una ROM o in una PROM. **Altri monitor realizzano dei breakpoint attraverso un effettivo controllo degli indirizzi e del contatore di programma, con metodi hardware o software.** Questo metodo permette di definire dei breakpoint anche in corrispondenza di locazioni che si trovano in una ROM o in una PROM, ma, essendo gli indirizzi controllati via software, rallenta l'esecuzione di un programma. In certi casi, è possibile anche indicare l'indirizzo a cui il processore deve trasferire il controllo al momento di un breakpoint. Un'altra possibilità è quella di un ritorno dipendente da un interruttore, come nell'esempio seguente.

```
BRKPT  BTST   #7,PIADR   ATTENDERE CHE L'INTERRUTTORE
       BNE    BRKPT      CORRISPONDENTE AL BIT 7
       RTE               VENGA CHIUSO
```

Naturalmente si possono usare altri dati o linee di controllo del PIA. Ricordate che RTE riabilita automaticamente gli interrupt.

Precauzioni nell'Uso dei Breakpoint

Quando si usano dei breakpoint (manualmente o tramite dei monitor) è opportuno prendere le seguenti precauzioni:

1. **Inserire dei breakpoint solo in corrispondenza di indirizzi che contengono dei codici operativi.** Sostituire dei dati o degli indirizzi con istruzioni Trap causa spesso confusione.
2. **Interpretare i risultati con molta attenzione.** Non dimenticate che il processore non ha ancora eseguito l'istruzione che è stata sostituita.
3. **Controllare tutte le varie condizioni prima di riprendere l'esecuzione.** Potrebbe essere necessario cambiare il contatore di programma, correggere il contenuto dei registri o delle locazioni di memoria, rimuovere dei breakpoint che non sono più necessari ed inserirne di nuovi. I metodi per riprendere l'esecuzione del programma variano notevolmente, perciò, a questo proposito, consultate la Guida Utente del vostro microcomputer. Fate attenzione a non riprendere l'esecuzione in mezzo ad un'istruzione (cioè, a un indirizzo che non contiene un codice operativo) oppure in mezzo ad operazioni di I/O o di temporizzazione (ad es., l'invio di dati ad una telescrivente), la cui esecuzione non può essere logicamente ripresa dopo un'interruzione.

DUMP DEI REGISTRI

Un metodo per il dump (visualizzazione del contenuto) dei registri consente di verificare i valori presenti in tutti i registri del processore o in alcuni in particolare. In genere, un dump dei registri fa parte della routine per la gestione dei breakpoint e del programma di debugging che controlla la funzione Trace.

Funzione del programma di dump

Un buon programma di dump consente di specificare quali registri, o anche quale porzione di determinati registri, visualizzare. Dal momento che con l'MC68000 è possibile operare selettivamente solo su un byte o una word di un registro, in certi casi, sarà sufficiente visualizzare, ad esempio, soltanto il byte meno significativo. Analogamente, se ci interessano solo alcuni registri dati, visualizzeremo semplicemente il loro contenuto e non quello di tutti i 16 registri dati ed indirizzi. **La Figura 19-2 mostra i risultati di un classico programma di dump dei registri.**

```
D0=3FD56709 D1=100002 D2=2430 D4=3C A0=00014000 A1=6000 A7=00056421
```

Figura 19-2. Risultati di un Tipico Dump dei Registri

C'è un paio di cose da tener presenti quando scriviamo un programma di questo tipo. Innanzitutto, se vogliamo visualizzare il contenuto del contatore di programma, sappiamo di poterlo trovare da qualche parte sullo stack. Tuttavia, bisogna tener conto delle Exception e/o

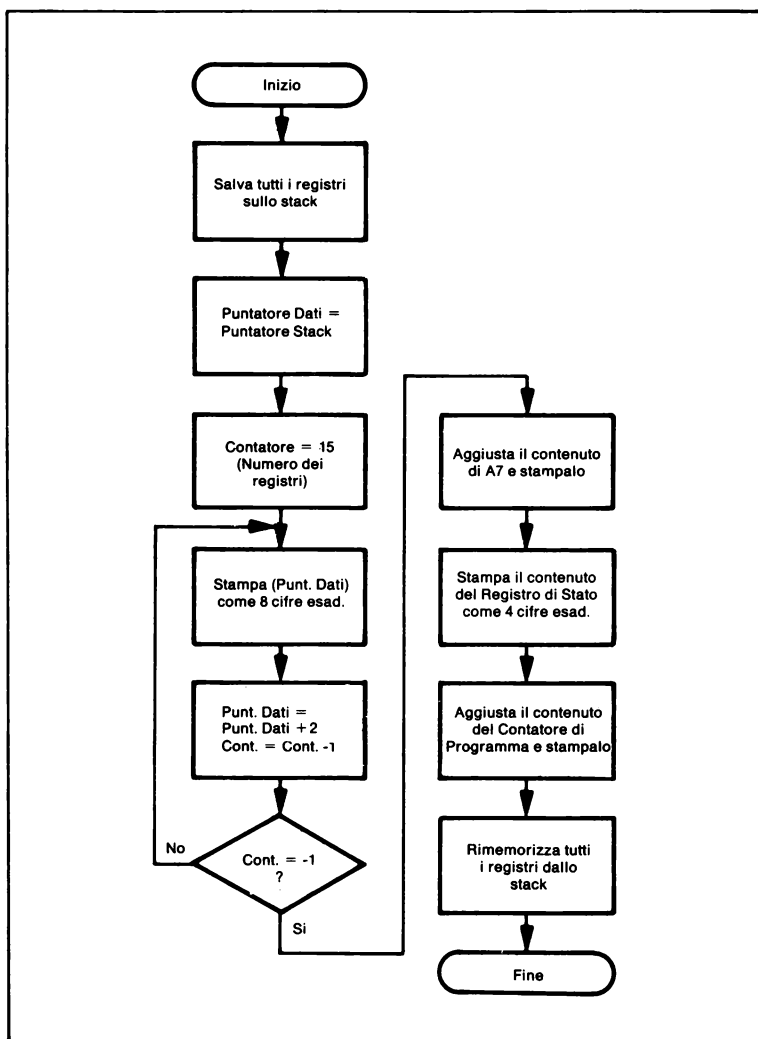



Figura 19-3. Diagramma di Flusso di un Programma per il Dump dei Registri.

chiamate di subroutine che hanno avuto luogo prima del programma di dump, dal momento che esse avranno depositato altri dati sullo stack.

In secondo luogo, il puntatore allo stack (A7) potrà darvi qualche problema, se non tenete conto del modo operativo in cui si trova il processore (Supervisore o Utente). Ecco alcune regole da ricordare:

- Nel modo Utente in A7 si trova il puntatore allo stack Utente e non è possibile disporre del puntatore allo stack Supervisore.
- Nel modo Supervisore in A7 si trova il puntatore allo stack Supervisore e si può ottenere il puntatore allo stack Utente con l'aiuto dell'istruzione MOVE USP,An.

Inoltre, non dimenticate che una chiamata di subroutine salva solo il valore del contatore di programma, mentre un'Exception (Trap, interrupt e così via), **oltre al contenuto del contatore di programma, salva anche quello del registro di stato.**

Infine, se vi trovate nel modo Utente e salvate da qualche parte il contenuto del registro di stato, non tentate, in seguito, di ripristinarne l'intero valore: si tratta di un'istruzione privilegiata. Sarà sufficiente ripristinare la parte relativa ai codici di condizione con un'istruzione MOVE in CCR. È disponibile, in alternativa, anche l'istruzione RTR, che ripristina automaticamente la parte del registro di stato che contiene i codici di condizione.

La figura 19-3 mostra il diagramma di flusso del programma per il dump dei registri, REGDUMP. Si presuppone che le subroutine PRT8HEX e PRT4HEX provvedano a convertire e visualizzare sulla stampante 32 o 16 bit del registro D0, in forma esadecimale. Si presuppone che la routine di dump venga richiamata con un'istruzione BSR o JSR e che il processore si trovi in modo Utente.

```
*
* PROGRAMMA PER IL DUMP DEI REGISTRI
*
PROGRAM EQU      $4000
        ORG      PROGRAM

REGDUMP MOVE.W   SR,-(A7)             SALVA IL REGISTRO DI STATO
        MOVEM.L  D0-D7/A0-A7,-(A7)    SALVA GLI ALTRI REGISTRI

        MOVEA.L  A7,A0                A0 E' UN PUNTATORE ALLO STACK LOCALE
        MOVEQ    #15-1,D4             SONO 15 I REGISTRI DA STAMPARE

LOOP    MOVE.L   (A0)+,D0             PRENDI IL REGISTRO
        BSR      PRT8HEX              E STAMPALO
        DBRA     D4,LOOP

        MOVE.L   (A0)+,D0             PRENDI IL PUNTATORE ALLO STACK
        ADDI.L   #6,D0                CORREGGI IL CONTENUTO
        BSR      PRT8HEX              STAMPALO

        MOVE.W   (A0)+,D0             PRENDI LA WORD DI STATO
        BSR      PRT4HEX              STAMPALA

        MOVE.L   (A0)+,D0             PRENDI IL VECCHIO CONT. DI PROGR
        SUBI.L   #2,D0                CORREGGI IL CONTENUTO
        BSR      PRT8HEX              STAMPALO

        MOVEM.L  (A7)+,D0-D7/A0-A7    RIPRISTINA I REGISTRI

        RTR                           RITORNA E RIPRISTINA I CODICI
                                      DI CONDIZIONE
        END      REGDUMP
```

Si noti che l'ultima istruzione è un RTR. Volendo richiamare il programma di dump attraverso un'Exception, saranno necessarie le modifiche indicate nel programma SYSDUMP.

```
*
* PROGRAMMA PER IL DUMP DEI REGISTRI DOPO UNA TRAP O UN'EXCEPTION
*
PROGRAM EQU      $4000
        ORG      PROGRAM

SYSDUMP MOVEM.L  D0-D7/A0-A6,-(A7)    SALVA I REG. SU STACK SUPERVISORE

        MOVEA.L  A7,A0                A0 E' UN PUNTATORE ALLO STACK LOCALE
        MOVEQ    *15-1,D4             SONO 15 I REGISTRI DA STAMPARE

LOOP    MOVE.L   (A0)+,D0             PRENDI IL REGISTRO
        BSR      PRT8HEX              E STAMPALO
        DBRA     D4,LOOP

        MOVE.L   USP,A1               PRENDI IL PUNT. ALLO STACK UTENTE
        MOVE.L   A1,D0
        BSR      PRT8HEX              STAMPALO
        MOVE.W   (A0)+,D0             PRENDI LA WORD DI STATO
        BSR      PRT4HEX              STAMPALA

        MOVE.L   (A0)+,D0             PRENDI IL VECCHIO CONT. DI PROGR.
        SUBI.L   *2,D0                CORREGGI IL CONTENUTO
        BSR      PRT8HEX              STAMPALO
        MOVEM.L  (A7)+,D0-D7/A0-A7    RIPRISTINA I REGISTRI

        RTE                           RITORNA E RIPRISTINA I CODICI
                                      DI CONDIZIONE
        END      SYSDUMP
```

Conoscete la differenza fra le istruzioni RTE, RTR e RTS? Quali sono quelle privilegiate? Perchè?

SINGLE-STEP

La disponibilità del single-step consente di eseguire un'istruzione o un ciclo di memoria per volta, visualizzando anche il contenuto di alcuni registri o locazioni di memoria. Di solito, questo sistema è associato con dei circuiti esterni che controllano le linee di output del processore. L'MC68000, tuttavia, dispone di circuiti interni che consentono l'esecuzione single-step, attraverso la sua logica Trace.

TRACE

La funzione Trace permette di verificare i risultati intermedi di un programma, dando la possibilità di conoscere i valori presenti nei registri del processore dopo ogni istruzione. In genere, **consente di specificare il numero di istruzioni da eseguire e di indicare la quantità ed il tipo di informazioni che devono essere visualizzate ad ogni arresto.** È possibile, inoltre, stampare il contenuto di determinate locazioni di memoria. In questo modo, possiamo selezionare solo quelle che ci sono veramente utili.

Limitazioni del trace

Tuttavia, **va tenuto presente che un'esecuzione di questo tipo rallenta notevolmente il processore.** Perciò, non c'e la possibilità di verificare dei cicli di ritardo o delle operazioni di I/O in tempo reale, nè di

individuare eventuali errori di temporizzazione o errori nei sistemi di interrupt e DMA. In realtà, l'esecuzione single-step avviene, di solito, ad una velocità pari ad un milionesimo di quella del processore. Un secondo del normale tempo di esecuzione corrisponderà a più di dieci giorni e, quindi, questo metodo **è utile solo per il controllo di brevi sequenze di istruzioni.**

Attivazione della funzione Trace

L'MC68000, a differenza della maggior parte dei microprocessori, ha una funzione Trace incorporata, che possiamo attivare mettendo a 1 il bit 15 del registro di stato. Quando il processore si trova in modo Trace, dopo ogni istruzione si verifica un'Exception che permette ad una routine di debugging di controllare l'esecuzione del programma. **Nel modo Trace è come se avessimo inserito dei breakpoints (sotto forma di Trap) dopo ogni istruzione.**

La risposta ottenuta è quella prevista per le Exception dovute ad istruzioni di tipo Trap. Il contenuto del contatore di programma e quello del registro di stato vengono salvati ed il controllo passa all'indirizzo contenuto nel relativo vettore di Exception, che, in questo caso, è il #9, alla locazione di memoria 24_{16}.

Volendo realizzare una semplice funzione Trace sul vostro sistema è sufficiente mettere l'indirizzo iniziale della routine per il dump dei registri (SYSDUMP), descritta in precedenza, nella locazione 24_{16} (quella del vettore di Exception per il modo Trace).

```
ORG   $24
DC.L  SYSDUMP
```

Quindi, bisogna porre a uno il bit 15 del registro di stato, usandc una delle opportune istruzioni, ed eseguire il programma. Dopo l'esecuzione di ciascuna istruzione, saranno visualizzati tutti i registri del processore (tranne il puntatore allo stack Supervisore). Per una descrizione più dettagliata del processo di Exception, vi rimandiamo ancora una volta al Capitolo 15. Non dimenticate che il valore del contatore di programma ottenuto con SYSDUMP deve essere modificato. In che modo?

Miglioramento della routine di Trace

Questa routine di Trace fornisce un gran numero di informazioni. Se desiderate migliorarla o se disponete già di un buon programma di Trace, ecco alcuni consigli utili:

1. **Stabilite, prima di tutto, quali sono le vostre necessità,** altrimenti non saprete come interpretare i risultati.

2. **Iniziate analizzando il comportamento di una o due variabili soltanto e senza stampare troppo spesso i risultati.** Un numero eccessivo di dati crea confusione.

3. **Utilizzate dei breakpoints per interrompere la funzione di Trace.**

4. **Utilizzate qualsiasi mezzo di cui dispone il vostro computer per definire gli output.** Altrimenti finirete per avere delle pagine piene di numeri senza un significato preciso e impiegherete un sacco di tempo solo per sapere di che tipo di dati si tratta.

5. **Fate attenzione quando richiedete** (se il vostro programma lo consente) **la visualizzazione parziale di un registro.** Situazioni particolari, come l'estensione del segno, possono causare dei problemi che non risulteranno evidenti, senza la visualizzazione dell'intero registro.

DUMP DELLA MEMORIA

Un programma per il dump della memoria invia il contenuto delle locazioni ad un dispositivo di output (ad es. una stampante). È un metodo molto efficace per esaminare matrici di dati o perfino interi programmi. Tuttavia, dump molto estesi non sono di grande utilità (servono solo a produrre cartacce), a causa della massa informe di dati visualizzati. Richiedono anche parecchio tempo, soprattutto con stampanti lente. Al contrario, **dei dump ridotti forniscono al programmatore una quantità ragionevole di informazioni che possono essere esaminate come un tutt'uno. Diventa facile, in questo modo, individuare il ripetersi di una sequenza di dati o gli offset di intere matrici.**

Scrivere un programma di dump non è facile. Inserite dei controlli per verificare che l'indirizzo finale dell'area di memoria, di cui si vogliono visualizzare i contenuti, non sia più piccolo di quello iniziale. Se così fosse, il programma dovrà segnalare l'errore oppure limitarsi a non fornire nessun tipo di output.

Dal momento che la velocità, con cui si ottiene un dump della memoria, dipende dalla velocità del dispositivo di output, l'efficienza della routine non è molto importante. **Il programma seguente ignora del tutto i casi in cui l'indirizzo iniziale è maggiore di quello finale ed opera su blocchi di memoria di lunghezza qualsiasi.**

```
                          *
                          * QUESTO PROGRAMMA STAMPA UNA PARTE DEL CONTENUTO
                          * DELLA MEMORIA
                          *
00006000:                 DATA     EQU      $6000
00004000:                 PROGRAM  EQU      $4000
                          ;
                                   ORG      DATA
00006000:                 START    DS.L     1
00006004:                 END      DS.L     1
                          ;
                                   ORG      PROGRAM
00004000: 2078 6000       MEMDUMP  MOVEA.L  START,A0     PRENDI IND. INIZIALE
00004004: 2278 6004                MOVEA.L  END,A1       PRENDI IND. FINALE
                          ;
00004008: B1C9            LOOP     CMPA.L   A1,A0        SE END > START
0000400A: 6206                     BHI.S    DONE         ...ALLORA DONE
0000400C: 2018                     MOVE.L   (A0)+,D0     ...ALTRIMENTI PRENDI DATO,
                          ;                              INCREMENTA START
0000400E: 6106                     BSR      PRT8HEX      E STAMPA IL DATO
00004010: 60F6                     BRA      LOOP
                          ;
00004012: 4E75            DONE     RTS
                                   END      MEMDUMP
```

Con questo programma si ottiene un risultato come quello mostrato nella Figura19-4. Dato che si tratta di long word, possiamo stampare un massimo di tre byte oltre l'indirizzo finale specificato. Per rendervene conto, provate con START = 6000 e END = 6004.

Questa routine funziona correttamente anche nel caso in cui la locazione iniziale e quella finale siano uguali (provate!). Se l'area di

dump comprende lo stack, i risultati vanno interpretati con molta attenzione, in quanto lo stack viene utilizzato anche dalla stessa routine di dump. Inoltre, anche la subroutine PRT8HEX può cambiare alcune locazioni della memoria e dello stack.

Chiaramente, non è facile interpretare questi risultati: non compaiono gli indirizzi ed il formato di output non è un granchè. **La Figura 19-5 mostra un formato migliore, che indica anche i relativi indirizzi e consente di distinguere facilmente byte, word e long word.**

Se lavorate molto con stringhe ASCII, allora vi sarà utile disporre dei caratteri ASCII contenuti nelle varie locazioni di memoria, come in Figura 19-6. È un formato di output molto comune e particolar-

```
48415353
45204D41
44452054
48495320
44554D50
```

Figura 19-4. Risultati di un Dump di Memoria non Formattato

```
005000  43  48  41  4C   4D  45  52  53   20  53  57  45   44  45  4E  20
```

Figura 19-5. Risultati di un Dump di Memoria Formattato

```
005000  54 48 45 20 4D 45 4D 4F 52 59 20 44 55 4D 50 20   THE.MEMORY.DUMP
```

Figura 19-6. Risultati di un Dump di Memoria con Caratteri ASCII

```
005000  54 48 45 20 4D 45 4D 15 4F 52 59 20 44 55 4D 50   THE.MEM.ORY.DUMP
```

Figura 19-7. Risultati di un Dump di Memoria ASCII con un Carattere non Stampabile

mente utile; ad esempio, vi fa immediatamente vedere se nella stringa è presente qualche carattere non stampabile.

Perciò, se abbiamo un byte con un valore 1516, fra la M e la O di MEMORY, il dump risulterebbe come quello mostrato nella Figura 19-7. Un programma di dump, in grado di visualizzare solo i caratteri stampabili, non lo avrebbe indicato.

Provate a riscrivere il programma per il dump della memoria, in modo da ottenere un output con i relativi indirizzi ed i valori esadecimali contenuti nelle varie locazioni, oltre ai corrispondenti caratteri ASCII.

STRUMENTI PIÙ COMPLESSI PER IL DEBUGGING

Altre tecniche di debugging molto utilizzate sono:

- **Programmi di simulazione per verificare la logica del programma**
- **Analizzatori logici per controllare i segnali e la temporizzazione**

Esistono molte varianti, ma noi ci limiteremo alla descrizione delle caratteristiche standard.

Simulatori

Il simulatore è l'equivalente computerizzato del vecchio sistema "carta e matita". **Si tratta di un programma che, passando attraverso le varie fasi del ciclo operativo di un computer, annota il contenuto di tutti i registri, dei flag e delle locazioni di memoria.** Naturalmente, sono tutte cose che potremmo fare manualmente, ma ciò riichiederebbe un notevole sforzo ed una estrema attenzione agli effetti di ogni singola istruzione. Un programma, invece, non si stanca mai, non rischia di confondersi, non dimentica un'istruzione o un registro e, soprattutto, non finisce mai la carta.

Le caratteristiche principali di un simulatore sono:

- **La possibilità di breakpoint.** I breakpoint possono essere previsti dopo che sono stati eseguiti un certo numero di cicli, quando si fa riferimento ad una locazione di memoria o ad un gruppo di locazioni oppure quando di queste locazioni si altera il contenuto.
- **Dump dei registri e della memoria,** in modo da poter visualizzare il contenuto delle locazioni di memoria, dei registri e delle porte di I/O.
- **Una funzione Trace,** per stampare il contenuto di particolari registri o locazioni di memoria, ogni volta che il programma li modifica o li utilizza.
- **La possibilità di definire i valori iniziali dei registri e/o delle locazioni di memoria** o di cambiarli nel corso della simulazione.

Alcuni simulatori sono in grado di simulare anche le operazioni di I/O, gli interrupt e il DMA. **I simulatori offrono molti vantaggi:**

1. Forniscono una descrizione completa dello stato del computer, dal momento che un simulatore non deve sottostare alle limitazioni dei piedini di uscita del microprocessore o di altre caratteristiche dei circuiti interni.
2. Consente di disporre di breakpoint, dump, Trace e di altre funzioni, senza utilizzare il sistema di controllo o la memoria del processore, in modo da non interferire con il programma da verificare.
3. Programmi, punti di partenza e altre condizioni sono facilmente modificabili.
4. Un progettista può disporre di tutte le caratteristiche di un grande computer, compreso il software e le periferiche.

Limitazione dei simulatori

D'altra parte, il simulatore presenta delle limitazioni, dal momento che si tratta pur sempre di una struttura software e, quindi, è separato dal microcomputer vero e proprio. Le limitazioni maggiori sono:

1. Il simulatore non permette di individuare eventuali problemi di temporizzazione, in quanto funziona ad una velocità inferiore rispetto a quella reale. Di solito è molto lento: sono necessarie molte ore per simulare tutte le operazioni che il microprocessore esegue in un secondo.
2. Il simulatore non fornisce un modello esatto della sezione di I/O, in quanto non è in grado di simulare con precisione l'hardware esterno o eventuali interfacce.

Il simulatore rappresenta il principale metodo software per il debugging di un programma ed ha, quindi, tutti i vantaggi e le limitazioni del caso. In genere, fornisce un gran numero di informazioni sulla logica del programma ed eventuali problemi relativi al software, ma spesso non è in grado di risolvere inconvenienti connessi alla temporizzazione, l'I/O e l'hardware.

Analizzatore Logico

L'analizzatore logico, o di microprocessori, è la soluzione hardware del debugging. Si tratta, sostanzialmente, di una versione digitale parallela di un normale oscilloscopio. Visualizza le informazioni in modo binario, esadecimale o mnemonico su un CRT e dispone di una varietà di eventi d'innesco, di valori soglia e di input. La maggior parte degli analizzatori dispone anche di una memoria, così da visualizzare i contenuti, anche quelli precedenti, dei bus del microprocessore.

Procedura standard di analisi

La procedura standard consiste nello stabilire un evento d'innesco, come la presenza di un determinato valore sul bus indirizzi o di una particolare istruzione sul bus dati. Ad esempio, si potrebbe

attivare l'analizzatore se il processore cerca di memorizzare un dato in una particolare locazione oppure tenta di eseguire un'istruzione di I/O ed osservare, poi, gli eventi che hanno preceduto il breakpoint. **In questo modo abbiamo la possibilità di individuare inconvenienti del tipo di brevi picchi di rumore (o glitches), errate sequenze di segnali, sovrapposizioni di forme d'onda ed altri errori di segnale o di temporizzazione. Naturalmente, non saremmo in grado di diagnosticare questo genere di errori con il solo aiuto di un simulatore software.**

Esistono vari tipi di analizzatori logici. Ecco in che cosa si differenziano:

- **Numero di ingressi.** Ne sono necessari almeno 40 per controllare un bus dati a 16 bit e un bus indirizzi a 24. Ce ne vogliono ancora di più se intendiamo verificare anche i segnali di controllo, i clock ed altri input particolari.
- **Disponibilità di memoria.** Ogni valore precedente, che viene salvato, occupa parecchi byte di memoria.
- **Frequenza massima.** Deve essere di parecchi MHz per gestire anche i processori più veloci.
- **Ampiezza minima del segnale** (importante per cogliere i "glitches").
- **Tipo e numero dei possibili eventi d'innesco.** Caratteristiche importanti sono gli intervalli pre- e postinnesco, che consentono all'utente di visualizzare eventi che si verificano prima e dopo quello d'innesco.
- **Metodi di collegamento al microcomputer.** Alcune volte è necessaria un'interfaccia piuttosto complessa.
- **Numero dei canali di visualizzazione.**
- **Visualizzazione in forma binaria, esadecimale o mnemonica.**
- **Formato della visualizzazione.**
- **Caratteristiche di stabilità dei segnali.**
- **Capacità della sonda.**
- **Soglie uniche o duplici.**

Tutti questi elementi sono importanti per confrontare i diversi analizzatori logici, dato che si tratta di strumenti nuovi e non standardizzati. Sono già disponibili moltissimi modelli ed altri se ne aggiungeranno in futuro.

Casi in cui è necessario un analizzatore

Naturalmente, **gli analizzatori logici sono necessari soltanto per quei sistemi caratterizzati da una temporizzazione molto complessa. I problemi hardware relativi ad applicazioni semplici, che utilizzano periferiche lente, possono benissimo essere individuati con un oscilloscopio tradizionale.**

DEBUGGING CON LISTE DI CONTROLLO

Nessuno può sperare di riuscire a controllare manualmente un intero programma; al massimo, possiamo verificare alcuni punti particolarmente critici. **Procedendo in modo sistematico, anche un**

semplice controllo manuale permette di individuare un gran numero di errori, senza far ricorso ad alcuno strumento di debugging.

Il problema è dove concentrare i nostri sforzi. La soluzione è di farlo in quei punti che possono essere chiariti con risposte di tipo sì-no o con un semplice calcolo aritmetico. Non cercate di fare dei calcoli troppo complessi, di seguire le variazioni dei flag di stato o di provare ogni possibile situazione. Limitatevi a quei problemi che possono essere risolti facilmente, affrontando, invece, gli aspetti più complessi con l'aiuto dei vari strumenti di debugging. Ma, soprattutto, sforzatevi di procedere in modo sistematico: realizzate delle liste di controllo e assicuratevi che il programma svolga correttamente tutte le operazioni fondamentali.

La prima fase consiste nel confrontare il diagramma di flusso, o un'altra eventuale documentazione, con il codice effettivo, accertandosi che vi sia un'esatta corrispondenza. Una semplice lista di controllo sarà sufficiente. È facile dimenticare un'intera diramazione o, addirittura, un'intera sezione di elaborazione.

Quindi, bisogna prendere in esame i loop, controllando che tutti i registri e le locazioni di memoria utilizzati siano inizializzati correttamente. Questa è una causa di errore piuttosto frequente; ancora una volta, basterà una semplice lista di controllo.

Esaminare tutti i salti condizionati, provando un caso in cui avviene una diramazione ed uno che, invece, non la dovrebbe provocare. La diramazione avviene correttamente o in modo errato? Se la diramazione avviene quando un numero è maggiore o minore di un altro, provate anche il caso in cui entrambi sono uguali. Il salto avviene nel modo giusto? Assicuratevi che la vostra scelta sia coerente con la definizione del problema.

Osservare i loop nel loro insieme, provando manualmente la prima e l'ultima iterazione: spesso sono questi i punti critici. Cosa accade se il numero delle iterazioni è zero (ad es., non ci sono dati o una tabella non contiene nessun valore)? L'esecuzione del programma continua correttamente? Spesso alcuni programmi eseguono un'iterazione, senza che questa sia necessaria, oppure, e questo è ancora peggio, decrementano i contatori al di sotto dello zero, prima di controllarli. Provate anche dei casi banali, in cui il programma non ha particolari alternative.

Controllare tutte le istruzioni fino all'ultima. Non vi illudete che il primo errore che trovate sia anche l'unico. Un preventivo controllo manuale vi aiuterà ad ottenere il massimo beneficio dalle esecuzioni di debugging, poichè, a quel punto, avrete già eliminato molti piccoli errori.

Domande da Porsi

Ecco un rapido elenco delle domande che ci dobbiamo porre effettuando un controllo manuale:

1. Il programma comprende tutto quanto è stato stabilito in fase di progettazione (e viceversa, per scopi di documentazione)?

2. I registri e le locazioni di memoria vengono tutti inizializzati prima di essere usati all'interno dei loop?
3. I salti condizionati sono corretti dal punto di vista logico?
4. Tutti i cicli iniziano e terminano in modo corretto?
5. I casi di identità sono gestiti correttamente?
6. I casi più banali sono gestiti correttamente?

RICERCA DEGLI ERRORI

Naturalmente, nonostante tutte queste precauzioni (e ancor più tralasciandone qualcuna), molto spesso un programma continuerà a non funzionare. Resta il problema di trovare gli altri errori. Gli elenchi che seguono possono esservi di aiuto. Abbiamo cercato di classificare tutti i possibili tipi di errori. Tuttavia, non dovete pensare che un certo errore si verifichi soltanto in un certo tipo di programma. La nostra classificazione ha solo lo scopo di permettervi di individuare più rapidamente un eventuale errore. Ma se questo non è indicato nella categoria in cui sembra più probabile doverlo trovare, provate a cercarlo nelle altre.

ERRORI PIÙ FREQUENTI IN DETERMINATE PARTI DEL PROGRAMMA

Sezione di Inizializzazione

- **Mancata inizializzazione di alcune variabili, come contatori, puntatori, somme, indici e così via.** Registri, locazioni di memoria e codici di condizione non contengono necessariamente degli zeri, prima di essere utilizzati. Assicuratevi anche di inizializzare la parte giusta di un registro. Se, ad esempio, intendete utilizzare il registro D0 come contatore ad 8 bit per un loop DBRA, è necessario azzerare l'intera word di ordine basso, dal momento che questa istruzione agisce sempre su un operando di una word (16 bit).

- **Errato funzionamento nei casi più banali.** È qui che si deve, di solito, decidere cosa fare se il programma non può fare niente (non ci sono dati, un elenco non contiene nessun valore e così via). Non presuppore che queste situazioni non si verificheranno, a meno che il programma non provveda esplicitamente ad eliminarle.

- **Inizializzazione accidentale.** Assicuratevi che nessuna istruzione di salto restituisca il controllo alla sezione di inizializzazione.

Loop

- **Aggiornare contatori, puntatori o indici al momento sbagliato o non aggiornarli affatto.** Assicuratevi che, in certi casi, non vengano saltate o ripetute più volte le istruzioni di aggiornamento. Fate particolarmente attenzione ai loop nidificati e ricordatevi che i contatori dei cicli più interni devono essere inizializzati ogni volta.
- **Confondere le operazioni di predecremento e postincremento.** Ricordate che il postincremento incrementa il registro indirizzi dopo averne usato il contenuto, mentre il predecremento lo decrementa prima di utilizzarlo. Non dimenticate, inoltre, che è la lunghezza dell'istruzione che stabilisce l'entità dell'incremento o del decremento. Con un'istruzione long word l'incremento o il decremento è di 4, con un'istruzione word è di 2 e con un'istruzione di un byte di 1. Avete indicato correttamente la dimensione?
- **Errato uso dell'istruzione DBcc.** La condizione indicata è quella che fa uscire il programma dal loop e non quella che provoca la diramazione all'interno del loop stesso. Se la condizione non viene soddisfatta, il processore decrementa il contatore e controlla se questo è uguale a -1: il test non viene eseguito con valori minori di zero. Inoltre, tenete conto del fatto che viene controllato il -1 (anzichè lo zero), altrimenti il ciclo sarà eseguito una volta in più del necessario.
- **Invertire la logica di un salto condizionato, effettuando, ad esempio, un salto con il Carry uguale a 1, mentre si intedeva ottenere la diramazione con il Carry uguale a zero.** Le istruzioni di confronto e sottrazione eseguono l'operazione *destinazione* (secondo operando) - *sorgente* (primo operando), assegnando gli opportuni valori ai flag di Carry e di Zero:

  Flag di Zero (Z) = 1 se destinazione < > sorgente
  Flag di Zero (Z) = 0 se destinazione > sorgente
  Flag di Carry (C) = 1 se destinazione < sorgente

  Il flag di Carry viene azzerato se destinazione = sorgente.
- **Modifica dei codici di condizione o loro mancata modifica.** L'istruzione MOVE influenza tutti i codici di condizione, tranne il flag di Extend (X). Le operazioni che utilizzano i registri indirizzi come operandi destinazione non modificano i codici di condizione, ad eccezione dell'istruzione CMPA. Quando ci sono più istruzioni che modificano i flag del registro di stato, servitevi anche delle indicazioni che vi abbiamo fornito con il Programma 9-2b.

Subroutine e Macro

- **Ignorare gli effetti delle subroutine e delle macro.** La chiamata di una subroutine o l'impiego di una macro provocano, di solito, l'esecuzione di numerose istruzioni che finiranno, quasi sempre,

per modificare il registro dei codici di condizione (CCR), oltre ad eventuali modifche di altri registri e di alcune locazioni di memoria. Assicuratevi di conoscere tutti gli effetti delle subroutine o delle macro che utilizzate. Ricordate anche quanto sia importante documentare le subroutine e le macro, affinchè un qualsiasi altro utente possa stabilire i loro effetti, senza essere costretto ad esaminare un lungo listato.

- **Dimenticare che lo stack viene utilizzato nel trasferimento del controllo alle subroutine.** Le istruzioni JSR e BSR salvano sullo stack l'indirizzo di ritorno, dopo eventuali altri parametri. L'istruzione RTS trasferisce semplicemente il controllo all'indirizzo che si trova alla sommità dello stack (Utente o Supervisore). Se non avete gestito correttamente lo stack, il processore potrebbe finire ad una locazione del tutto imprevista.

- **Utilizzare l'istruzione di ritorno sbagliata.** RTS non ripristina i codici di condizione, come, invece, fa RTR. Nessuna delle istruzioni per la chiamata di una subroutine salva automaticamente il contenuto dei codici di condizione: tocca a voi assolvere questo compito. Non dimenticate che RTR preleva i codici di condizione, prima di ripristinare il contatore di programma; perciò, la sequenza

```
MOVE.W SR,-(A7)
BSR    SUBR
```

  non funzionerà con un'istruzione RTR. Invece, se volete salvare i codici di condizione, dovete farlo all'inizio della subroutine, cui viene trasferito il controllo.

- **Mancato ripristino dei registri precedentemente salvati.** È un errore molto comune. Assicuratevi di ripristinare il numero corretto di registri e di usare le locazioni giuste. Servitevi dell'istruzione MOVEM.L per salvarli sullo stack. Non dimenticate che se spostate delle word da 16 bit dalla memoria a dei registri indirizzi, esse verranno traformate in valori a 32 bit mediante l'estensione del segno e questo può causare dei problemi.

- **Un uso improprio delle istruzioni Link ed Unlink.** Non cambiate il "registro di link", durante l'esecuzione di una subroutine. Ad esempio, usando LINK A6,#-16 all'inizio di una subroutine, A6 deve avere esattamente lo stesso valore quando viene eseguita l'istruzione UNLK A6. Altrimenti lo stack resterà sfasato, provocando un risultato disastroso per l'intero sistema. Inoltre, lo spostamento viene interpretato come un intero in complemento a due; se avete uno stack che cresce verso il basso (come fa lo stack di sistema), sarà necessario specificare uno spostamento negativo con l'istruzione LINK. Lo spostamento deve, inoltre, essere sempre un numero pari, dal momento che lo stack è organizzato sotto forma di word successive.

- **Invertire l'ordine degli operandi.** L'istruzione MOVE D1,D2 sposta il contenuto di D1 in D2. (Questo è esattamente l'opposto di ciò che accade con lo Z8000 e l'8086). Ricordate anche che SUB sorg.,dest. e CMP sorg.,dest. eseguono un'operazione del tipo dest. - sorg. L'istruzione DIV sorg.,Dn esegue l'operazione Dn/sorg. (e salva il risultato in Dn).
- **Confondere i modi di indirizzamento**
  - **Dati con indirizzi** (immediati ed assoluti). MOVE.W #$2000,D0 carica nel registro D0 il numero 2000_{16}, mentre MOVE.W $2000,D0 carica nel registro D0 il contenuto delle locazioni di memoria 2000_{16} e 2001_{16}.
  - **Diretto e indiretto a registro indirizzi.** Ricordate che CLR.L A0 carica degli zeri nel registro A0, mentre CLR.L (A0) mette degli zeri nella word di memoria indicata da A0.
  - **Dimenticare che i modi di indirizzamento funzionano diversamente con le istruzioni di salto.** Le istruzioni JMP e JSR sono eseguite con un livello di "indirettezza" in meno. Ad esempio, JMP $1000 pone 1000_{16} nel contatore di programma, mentre MOVE $1000,A0 mette il contenuto della locazione di memoria 1000_{16} nel registro D0.
- **Trascurare il fatto che certe istruzioni operano solo con un certo formato.**

  Esempi: DBcc sottrae 1 dai 16 bit di ordine basso del registro dati specificato.

  MOVEQ agisce su tutti i 32 bit del registro dati indicato.

  MOVE ea,-(A7) e MOVE (A7)+,ea devono essere sempre utilizzate con un indirizzo pari (inizio di una word).

  MOVE a CCR è un'istruzione di una word, ma interessa solo il byte di ordine basso del registro di stato.

  DIVS e DIVU influenzano tutti i 32 bit del registro dati destinazione, ma usano solo 16 bit dell'operando sorgente. La stessa cosa vale per MULS e MULU.

  Quando un registro indirizzi è usato come operando destinazione, è l'intero registro che viene interessato, indipendentemente dalle dimensioni indicate. Se è specificato che l'operando sorgente è una word, esso viene trasformato in un valore a 32 bit, mediante l'estensione del segno, nel registro indirizzi.
- **Dimenticare che l'MC68000 trasforma gli indirizzi a 16 bit in valori a 32 bit mediante l'estensione del bit di segno.** Questo può provocare dei problemi, lavorando nello spazio di memoria compreso fra 32 e 64 K (indirizzi da 8000_{16} a $FFFF_{16}$). Fate molta attenzione quando caricate dei valori immediati in un registro indirizzi e quando usate l'indirizzamento assoluto corto. In entrambi i casi, si possono ottenere degli strani risultati con dimensioni pari ad una word e se il bit più significativo della word è 1: l'estensione automatica del segno metterà degli 1 in tutti i 16 bit più significativi di un indirizzo long word.

- **Dimenticare i dettagli della estensione del segno con i dati.** MOVEQ considera l'operando come un valore provvisto di segno, che provvede ad estendere. ADDQ e SUBQ funzionano solo con i numeri positivi. MOVEM effettua l'estensione del segno, quando trasferisce delle word dalle locazioni di memoria ai registri.
- **Usare impropriamente le istruzioni di shift.** Ricordate le differenze fra gli shift aritmetici, logici e le istruzioni di rotazione. Essi influenzano i codici di condizione, anche se agiscono su un dato di una locazione. Utilizzando un registro dati per indicare il numero di shift, non dimenticate che questo valore è interpretato in modulo 64.
- **Confondere valori ad 8, 16 e 32 bit.** Il processore non si preoccupa di annotare se la variabile che avete messo in un registro era un valore ad 8, 16 o 32 bit. Tocca a voi specificarne le dimensioni in ciascuna istruzione. Ecco, a questo proposito, alcune cose da tenere bene in mente:
  - Un byte può contenere due numeri BCD e le istruzioni BCD (ABCD,SBCD) hanno una dimensione di un byte.
  - Una word di 16 bit occupa due byte e, quindi, "due indirizzi di memoria". In altri termini, una word da 16 bit, memorizzata nella locazione 1000_{16}, occupa anche la locazione 1001_{16}.
  - Una long word (32 bit) occupa 4 byte; questo può causare spesso degli errori, se siete abituati ad un microprocessore ad 8 bit.

- **Ignorare le limitazioni della memoria di sola lettura.** Evidentemente, le istruzioni che leggono e scrivono nelle locazioni di memoria hanno poco senso se applicate ad un indirizzo occupato da un dispositivo con memoria di sola lettura (ROM). Un programma di ordinamento, al quale siano stati forniti dei dati posti nella ROM, continuerà all'infinito!
- **Servirsi del registro sbagliato.** L'MC68000 dispone di un gran numero di registri dati e indirizzi. Mentre, da una parte, questo è uno dei motivi della potenza e della flessibilità del processore, dall'altra richiede una particolare attenzione. L'indicazione di due registri dati può differire di un solo carattere (ad es. D1, D2). La stessa cosa vale anche fra i registri dati e i registri indirizzi (ad es. A1, D1). Gli errori di battitura sono frequenti e difficili da individuare.
- **Confondere numeri BCD, binari, esadecimali e decimali.** Nella rappresentazione BCD, ogni cifra decimale è codificata separatamente in forma binaria, usando quattro cifre binarie (0 o 1). Nella rapresentazione esadecimale, quattro cifre binarie sono raggruppate insieme e rappresentate mediante una cifra esadecimale (da 0 a F). Ad esempio, il numero decimale 54_{10} è uguale a 110110_2 in binario, a 36_{16} in esadecimale e a 54_{16} nella rapresentazione BCD standard.
- **Dimenticare di trasferire il controllo al di là di sezioni di programma che non devono essere eseguite.** Il processore procede in modo sequenziale, a meno che non gli sia indicato di fare altrimenti. In

certi casi, sono necessari dei salti non condizionati per evitare routine che non devono essere eseguite.

- **Confondere lo stack ed i suoi puntatori.** Il contenuto dello stack è sempre indirizzato in uno dei modi indiretti ed il puntatore nel modo diretto a registro.
- **Confondere le posizioni dei bit nelle istruzioni che agiscono sui bit.** I bit sono numerati da 31 a 0. In un byte il bit meno significativo è lo zero, mentre il 7 è quello più significativo (MSB); il bit 15 è l'MSB di una word, il bit 31 lo è di una long word.

Errori nella manipolazione delle Stringhe

- **Determinare in modo errato la lunghezza di un array. Non dimenticate che gli indirizzi compresi fra 1000_{16} e 1004_{16} comprendono cinque (non quattro) locazioni di memoria. Così il numero degli elementi di un array è uguale a** *indirizzo finale - indirizzo iniziale + 1.*
- **Confondere numeri e caratteri.** La rappresentazione ASCII di una cifra non è uguale alla rappresentazione binaria o BCD. Ad esempio, la rappresentazione ASCII del numero sette è 37_{16}; 07_{16} è il carattere ASCII BELL, che fa suonare il campanello di una telescrivente.
- **Dimenticare che operazioni di tipo word non funzionano con indirizzi dispari.** Le operazioni sulle stringhe sono spesso delle dimensioni di un byte. Fate attenzione ad usare operazioni word o long word per spostare delle stringhe o aggiungere dei caratteri ad una stringa. Ad esempio, se il registro A4 contiene l'indirizzo della posizione attuale all'interno di una stringa e volete aggiungere il testo "The End", la sequenza di istruzioni

```
MOVE.L #"THE",(A4)+
MOVE.L #"END",(A4)+
```

provocherà una Exception per errore di indirizzo, se, prima dell'esecuzione della prima istruzione, A4 indica un indirizzo dispari.

Errori di Input/Output

- **Ignorare le limitazione fisiche dell'I/O e dei chip d'interfacciamento.** Mentre noi indirizziamo i chip d'interfacciamento, come se fossero delle locazioni di memoria, essi possono non comportarsi come tali. Salvare dei dati in una porta di input ha poco senso, come, del resto, caricare dei dati da una porta di output, a meno che questa non sia dotata di latch e di buffer. Alcuni dispositivi di I/O hanno due registri diversi (uno di sola lettura ed uno di sola scrittura) corrispondenti allo stesso indirizzo. I registri di stato e di controllo dell'ACIA 6850 ne sono un esempio. Fate attenzione alle istruzioni di shift, negazione e così

via, che leggono e subito riscrivono nella "stessa" locazione, provocando degli strani errori nel caso di combinazioni di registri del tipo di quelle presenti nel 6850.

- **Usare bit sbagliati dei registri di controllo e di stato.** L'ordine dei bit in questo tipo di registri può apparire casuale. Siete certi di aver usato la combinazione giusta?
- **Un uso errato dell'istruzione MOVEP.** Ricordate che questa istruzione utilizza indirizzi alternati, tutti pari o tutti dispari.
- **Dimenticare di eseguire il reset o l'inizializzazione dei dispositivi di I/O.** Ad esempio, l'ACIA 6850 richiede una sequenza di reset via software.

Errori Relativi all'Assemblatore

L'uso di un assemblatore è il solo modo pratico di convertire un programma sorgente in codice oggetto, ma può essere causa di alcuni inconvenienti fastidiosi. In particolare,

Operazioni di default dell'assembler dell'MC68000

- **Fate attenzione a ciò che l'assemblatore adotta come opzioni di default. Ad esempio, l'assemblatore standard per l'MC68000 utilizza le seguenti:**
  - **Il valore di default per la lunghezza di un'istruzione è di una word, a meno che non sia specificato diversamente.** È buona abitudine indicare sempre le dimensioni di un'istruzione, anche se, evidentemente, non sarebbe necessario con istruzioni della lunghezza di una word.
  - **I numeri, che non sono preceduti da alcun segno, vengono considerati decimali.** Se volete dei numeri esadecimali, dei caratteri ASCII e così via, dovete indicarli esplicitamente.
  - **I modi di indirizzamento di default sono quello diretto a registro e quello assoluto.** Cioè, A1 indica il registro indirizzi A1, non la locazione di memoria il cui indirizzo si trova nel registro. Il valore $1000 indicherà la locazione di memoria 1000_{16} e #$1000 indicherà il numero 1000_{16}.
- **Fate attenzione agli indirizzi assoluti corti.** Se avete usato la direttiva ORG, l'assemblatore presuppone che ogni riferimento ad un indirizzo assoluto corto richieda l'uso della forma dell'istruzione propria, appunto, dell'indirizzamento assoluto corto. Il processore, poi, estenderà il segno di questo indirizzo. Alcune versioni successive dell'assemblatore hanno ovviato a questo inconveniente.
- **L'assemblatore sceglie la forma rapida di un'istruzione ogni volta che questo è possibile, anche se non è stato specificato.** Perciò, ADD #2,D0 produrrà il codice oggetto corrispondente all'istruzione ADDQ.
- **Attenzione agli errori di battitura. I numeri dei registri sono vicini l'uno all'altro sulla tastiera e nessun assemblatore può accorgersi di un errore di battitura, qualora il risultato dello sbaglio sia un'istruzione valida.** Inoltre, alcuni assemblatori rimangono confusi dall'inserimento di spazi dove non si aspettano di trovarli o

dall'impiego accidentale di caratteri senza senso, come 1/2. In effetti, l'assemblatore, in certi casi, rileva anche errori in fin dei conti trascurabili, per poi accettare inserimenti del tutto illogici, ma non previsti dal suo realizzatore.

- **Ricordate che l'assemblatore può stampare un messaggio del tutto rassicurante, come NUMERO DI ERRORI 0, anche quando il programma è sbagliato. Questo messaggio significa soltanto che l'assemblatore non ha trovato errori secondo la sua interpretazione delle regole del linguaggio. Questo non esclude eventuali errori che danno istruzioni valide o che vanno oltre la comprensione dell'assemblatore, e, sopratutto, non esclude la possibilità di errori logici nel programma e non significa necessariamente che questo sia in grado di fare ciò che avevate previsto.**

Exception

L'elaborazione delle Exception, dal punto di vista degli eventuali errori, può essere suddivisa in due gruppi: gli interrupt e tutti gli altri tipi di Exception. Questo è dovuto al fatto che, in genere, gli interrupt sono controllati da dispositivi esterni e, quindi, sono eventi del tutto casuali. Gli altri tipi di Exception, istruzioni illegali, errori di indirizzi e così via, corrispondono spesso a una sequenza di istruzioni o ad una singola istruzione. Se il vostro microcomputer non dispone di un sistema software per l'elaborazione delle Exception, sarebbe molto utile scriverne uno, che dica, almeno, quale Exception ha provocato una Trap (errore di indirizzo, errore di bus e così via) e fornisca l'indirizzo dove si è verificata.

Alcuni possibili errori relativi all'uso delle Exception di qualsiasi tipo sono:

- **Dimenticare le caratteristiche delle Exception.** Il processore viene messo in modo Supervisore ed alcune informazioni (di solito il contatore di programma ed il registro di stato) sono salvate sullo stack Supervisore.
- **Usare l'indirizzo di ritorno sbagliato.** RTE e RTR non sono la stessa cosa, per cui non si possono utilizzare le varie subroutine così come sono, durante la fase di elaborazione di un'Exception. RTE ripristina l'intero registro di stato, mentre RTR ripristina soltanto la parte relativa ai codici di condizione. RTE è un'istruzione privilegiata.
- **Errori multipli di bus o di indirizzo.** Se il processore riconosce un errore di bus o di indirizzo, mentre ne sta già elaborando un altro, cessa l'esecuzione del programma (Halt). Ad esempio, supponiamo che, per una ragione qualsiasi, abbiate un valore dispari nel puntatore allo stack Supervisore. Quando usate il puntatore con questo valore dispari, si verificherà una Trap di indirizzo. Ma anche il sistema di gestione delle Trap utilizza lo stack Supervisore e questo provoca un nuovo errore di indirizzo, che provocherà l'arresto del processore.

Programmi Gestiti da Interrupt

Il debugging dei programmi gestiti mediante interrupt è particolarmente difficile, dal momento che eventuali errori possono manifestarsi solo se un interrupt si verifica in un momento particolare. Se, ad esempio, il programma abilita gli interrupt alcune istruzioni troppo presto, l'errore apparirà evidente solo nel caso che si verifichi un interrupt mentre il processore sta eseguendo quelle poche istruzioni. In generale, si può supporre che errori sporadici o casuali siano da attribuire al sistema di interrupt.

Dato che l'MC68000 ha una maschera di priorità degli interrupt nel registro di stato, è possibile eliminare alcuni degli interrupt ed individuare l'errore. Alcune volte un breakpoint messo all'inizio di una routine di interrupt può fornire un'indicazione sulle cause del problema, benchè questo sia un pò improbabile nei sistemi in tempo reale. Un altro metodo è quello di salvare gli indirizzi di ritorno ogni volta che si verifica un interrupt, localizzando, in questo modo, la parte del programma che causa l'errore.

Individuazione degli errori legati agli interrupts

Ecco alcuni degli errori più frequenti nei programmi gestiti tramite interrupt:

- **Un errato valore del livello di priorità degli interrupt.** Quando si esegue un reset del processore, la maschera di priorità degli interrupt viene messa al livello 7. Al momento del riconoscimento di un interrupt, ad essa viene assegnato un valore uguale al livello dell'interrupt appena ricevuto. RTE servirà a ripristinare, oltre al registro di stato, anche il livello di priorità degli interrupt. Assicuratevi che il programma ponga il livello di priorità degli interrupt al valore desiderato.
- **Abilitare interrupt di un certo livello, prima che il sistema sia in grado di gestirli.** I parametri del sistema, come i codici di condizione, i flag, i puntatori ed i contatori, devono essere prima inizializzati. In questo caso, una lista di controllo potrebbe essere di valido aiuto.
- **Dimenticarsi di salvare e ripristinare dei registri.** Gli interrupt sono un pò come delle subroutine. Bisogna usare le stesse precauzioni, quando si salvano e si ripristinano i registri o si riserva dello spazio sullo stack.
- **Dimenticare che un interrupt lascia il contenuto del vecchio contatore di programma e del registro di stato sullo stack, sia che li usiate o no.**
- **Dimenticare di azzerare la sorgente dell'interrupt, prima di uscire dalla relativa routine di servizio.** Ad esempio, se l'interrupt proviene da un PIA la routine di servizio deve leggere il registro dati del PIA, allo scopo di azzerare il flag di interrupt. L'operazione di lettura è necessaria anche se l'interrupt proviene da un dispositivo di output o da un clock in tempo reale; altrimenti, esso resta attivo e sarà riconosciuto di nuovo, non appena il processore riabilita gli interrupt di quel livello.

- **Non disabilitare determinati interrupt, durante trasferimenti multiword o altre sequenze critiche.** Ad esempio, supponiamo di avere un clock in tempo reale con sei cifre memorizzate in sei byte di memoria consecutivi. Se il clock contiene 115959 e le cifre vengono lette dalla memoria una per volta, senza disabilitare gli interrupt che aggiornano il clock, ecco ciò che può accadere. Se si verifica un interrupt dopo che è stata letta la seconda cifra, esso farà sì che le ultima quattro cifre siano 0000 e risulterà un valore 110000. Un errore di questo genere è difficile da trovare, perchè si verifica molto raramente e solo in presenza di vari fattori concomitanti.
- **Non riabilitare gli interrupt, dopo avere eseguito una routine che ne prevede l'inattivazione.**
- **Ignorare la possibilità che la routine di interrupt sia richiamata dal suo interno,** al pari di una qualsiasi subroutine (cfr. Capitolo 11).

Metodo "alternativo" per l'individuazione degli errori

Una lista dei possibili errori può essere infinita e lo scopo dell'elenco precedente è solo quello di fornire alcuni suggerimenti riguardo al punto da cui iniziare la ricerca. Sfortunatamente, nessuno ha trovato l'algoritmo che indichi come essere sicuri al cento per cento di aver trovato tutti gli errori. È difficile individuare tutti gli errori anche adottando dei metodi di ricerca sistematici. Qualche volta il metodo seguente è quello che dà i risultati migliori: spegnere il computer, bere una birra e far riposare il cervello. Magari abbandonare il problema per una notte o farlo esaminare da un'altra persona, da un diverso punto di vista. Spesso, spiegando il problema a qualcun'altro, riusciamo a trovarne la soluzione da soli.

ESEMPI DI PROGRAMMAZIONE

19-1. DEBUGGING DI UN PROGRAMMA DI CONVERSIONE

Lo scopo di questo programma è di convertire una cifra decimale, nella locazione di memoria DIGIT, in un codice a 7 segmenti e salvarlo nella locazione CODE. Il programma deve ripulire il display se DIGIT non contiene una cifra decimale. Sembra essere un compito piuttosto semplice. Sulla base del diagramma di flusso mostrato in Figura 19-8, il nostro primo tentativo di codifica sarà:

Programma Iniziale: (dal diagr. di flusso della figura 19-8).

```
*
* CONVERSIONE BCD - SETTE SEGMENTI
*
* INPUT -- CIFRA BCD
* OUTPUT -- CODICE SETTE SEGMENTI
*
DATA    EQU     $8000
PROGRAM EQU     $4200

        ORG     DATA

DIGIT   DS.B    1
CODE    DS.B    1
```

```
SSEG      DC.B     $3F,$06,$5B,$4F,$66
          DC.B     $6D,$7F,$07,$7F

BCD_7SEG MOVEA.W  SSEG,A0             PRENDI IND. DI BASE DELLA TABELLA
          MOVE     DIGIT,D0            PRENDI LA CIFRA DA CONVERTIRE
          CMP.B    #9,D0               SE MAGGIORE DI 9
          BCS.S    DONE                ALLORA DONE

          EXT.W    D0                  ALTRIMENTI ESTENDI IL SEGNO
          MOVE.B   0(A0,D0),D1         PRENDI CODICE DALLA TABELLA

          MOVE.B   D0,CODE

DONE      RTS

          END      BCD_7SEG
```

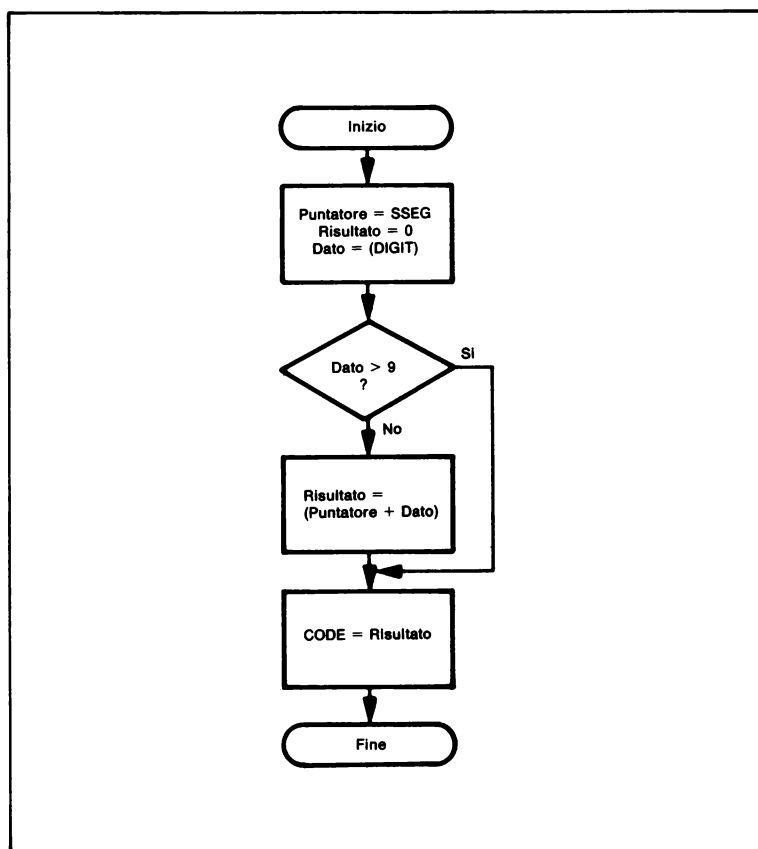



Figura 19-8. Diagramma di Flusso per una Conversione da Decimale a Sette Segmenti.

Uso delle Liste di Controllo

Per valutare questo programma., facciamo uso delle liste di controllo, che abbiamo descritto in precedenza,

1. Nel programma sono presenti tutti gli elementi del progetto iniziale? No! Abbiamo dimenticato la sezione che azzera il display, quando il dato non è una cifra decimale.

2. Inizializzazione? Okay!
3. I salti condizionati sono corretti? No! BCS (Branch on Carry Set) non funzionerà correttamente in caso di uguaglianza. (Provate!). L'istruzione giusta è BHI.S DONE.
4. Non ci sono loop.
5. Casi di uguaglianza? Sì, adesso sono gestiti in modo corretto.
6. Casi banali? Sì, (DIGIT) = 0 è gestita allo stesso modo di altre cifre. (Dato che zero è solo un'altra cifra , questo non è in realtà un caso particolare).

Abbiamo dimenticato anche di specificare il suffisso per la seconda istruzione MOVE. La seconda versione del nostro programma sarà:

Secondo Programma:

```
DATA     EQU      $8000
PROGRAM  EQU      $4200

         ORG      DATA

DIGIT    DS.B     1
CODE     DS.B     1
SSEG     DC.B     $3F,$06,$5B,$4F,$66
         DC.B     $6D,$7F,$07,$7F

BCD_7SEG MOVEA.W  SSEG,A0                PRENDI IND. DI BASE DELLA TABELLA

         MOVEQ    #0,D1
         MOVE.B   DIGIT,D0               PRENDI LA CIFRA DA CONVERTIRE
         CMP.B    #9,D0                  SE MAGGIORE DI 9
         BHI.S    DONE                   ALLORA DONE

         EXT.W    D0                     ALTRIMENTI ESTENDI IL SEGNO
         MOVE.B   0(A0,D0.W),D1          PRENDI CODICE DALLA TABELLA

         MOVE.B   D0,CODE

DONE     RTS

         END      BCD_7SEG
```

Il controllo manuale non ha rivelato nessun errore in questa versione.

Assemblaggio

La fase successiva consiste nel battere il programma ed assemblarlo.

Terzo Programma:

```
00008000:                  DATA     EQU      $8000
00004200:                  PROGRAM  EQU      $4200
                           ;
                                    ORG      DATA

00008000:                  DIGIT    DS.B     1
00008001:                  CODE     DS.B     1
00008002:                  SSEG     DC.B     $3F,$06,$5B,$4F,$66
00008007:                           DC.B     $6D,$7F,$07,$7F
                           ;
                                    ORG      PROGRAM

00004200: 3079 0000
00004204: 8002             BCD_7SEG MOVEA.W  SSEG,A0      PRENDI IND. DI BASE DELLA TABELLA
00004206: 7200                      MOVEQ    #0,D1
00004208: 1039 0000
0000420C: 8000                      MOVE.B   DIGIT,D0     PRENDI LA CIFRA DA CONVERTIRE
0000420E: 0C00 0009                 CMP.B    #9,D0        SE MAGGIORE DI 9
00004212: 620C                      BHI.S    DONE         ALLORA DONE
                           ;
```

```
00004214: 4880                    EXT.W    D0              ALTRIMENTI ESTENDI IL SEGNO
00004216: 1230 0000               MOVE.B   0(A0,D0.W),D1   PRENDI IL COD. DALLA TABELLA
                          ;
0000421A: 13C0 0000
0000421E: 8001                    MOVE.B   D0,CODE
                          ;
00004220: 4E75            DONE    RTS

                                  END      BCD_7SEG
```

Single Step

È giunto il momento di eseguire il programma istruzione per istruzione (single-step): sarà una cosa piuttosto rapida dato che si tratta di un programma breve. Se avete la possiobilità di specificare i registri che devono essere mostrati, dopo ogni istruzione, scegliete PC, D0, D1, SR ed A0.

Abbiamo scelto i seguenti dati di prova:

0 La cifra decimale più piccola
9 La cifra decimale più grande
10 Un caso limite
6B Un valore scelto a caso

Per il primo tentativo mettiamo uno zero nella locazione di memoria DIGIT. Dopo aver eseguito la prima istruzione, MOVE.W SSEG,A0, troviamo il valore $3F06_{16}$ nel registro A0. Non sembra giusto: ci saremmo aspettati di trovare il valore 8000_{16}. La prima cosa da controllare è se abbiamo utilizzato il tipo di indirizzamento corretto. In questo caso, la risposta è no: abbiamo confuso l'indirizzamento immediato con quello assoluto. Sostituite SSEG con #SSEG e provate di nuovo. Questa volta otteniamo $FFFF8000_{16}$ nel registro A0. Di nuovo, non si tratta del risultato che avevamo previsto. Tuttavia, tutte le F sono da attribuire alla estensione del segno (ammesso che A0, all'inizio, contenesse zero). Abbiamo specificato che in A0 deve essere caricato un indirizzo della lunghezza di una word, il cui bit più significativo è 1. Quando questo indirizzo subisce l'estensione del segno, verranno messi tutti '1' (FFFF) nella word più significativa del registro. La soluzione è quella di indicare la forma long word dell'istruzione MOVEA. Tutti questi guai ci sono serviti per capire che l'istruzione corretta è MOVEA.L #SSEG,A0. (ancora una volta, questo è un incoveniente dovuto al tipo di assemblatore che stiamo usando e, probabilmente, sarà eliminato nelle versioni successive).

Dopo questa prima fase, esaminiamo la parte restante del programma. Tutto sembra funzionare nel modo giusto. La diramazione non avviene e, come previsto, otteniamo $3F_{16}$ nel registro D1. Ma quando controlliamo la locazione di memoria CODE, troviamo la cifra BCD (zero) che abbiamo impiegato come dato di prova. Qualcosa non va. Per definire il problema, ci dobbiamo domandare: quali istruzioni agiscono sulla locazione di memoria CODE? In questo caso, si tratta dell'ultima istruzione: MOVE D0,CODE. Che cosa abbiamo nel registro D0? Il codice BCD. Dove si trova il codice a 7 segmenti? Nel registro D1. Aha! È probabile che si tratti di un

errore di battitura. Cambiamo D0, CODE in D1, CODE e facciamo un altro tentativo. Questa volta troviamo $3F_{16}$ nella locazione di memoria CODE. Il programma allora sarà:

Quarto Programma

```
00008000:                 DATA      EQU      $8000
00004200:                 PROGRAM   EQU      $4200
                          ;
                                    ORG      DATA

00008000:                 DIGIT     DS.B     1
06008001:                 CODE      DS.B     1
00008002:                 SSEG      DC.B     $3F,$06,$5B,$4F,$66
00008007:                           DC.B     $6D,$7F,$07,$7F
                          ;
                                    ORG      PROGRAM

00004200: 207C 0000
00004204: 8002            BCD_7SEG  MOVEA.L  #SSEG,A0          PRENDI IND. DI BASE DELLA TABELLA
00004206: 7200                      MOVEQ    #0,D1
00004208: 1039 0000
0000420C: 8000                      MOVE.B   DIGIT,D0          PRENDI LA CIFRA DA CONVERTIRE
0000420E: 0C00 0009                 CMP.B    #9,D0             SE MAGGIORE DI 9
00004212: 620C                      BHI.S    DONE              ALLORA DONE
                          ;
00004214: 4880                      EXT.W    D0                ALTRIMENTI ESTENDI IL SEGNO
00004216: 1230 0000                 MOVE.B   0(A0,D0.W),D1     PRENDI IL COD. DALLA TABELLA
                          ;
0000421A: 13C1 0000
0000421E: 8001                      MOVE.B   D1,CODE
                          ;
00004220: 4E75            DONE      RTS

                                    END      BCD_7SEG
```

Esecuzione di Prova

Questa volta eseguiamo l'intero programma con il secondo dato di prova, 9. Nella locazione di memoria CODE non troviamo $7D_{16}$, che è l'ultimo valore nella tabella del codice a 7 segmenti. Con un valore di input uguale a 9 il programma dovrebbe seguire un andamento analogo a quello con input uguale a zero. Per capire che cosa è accaduto, proviamo con un'altra esecuzione single-step. Tutto funziona bene finchè non arriviamo all'istruzione MOVE.B 0(A0,D0.W),D1. Ci attendevamo che $7D_{16}$ fosse caricato in D1, ma questo non è avvenuto. Un dump della memoria in corrispondenza della tabella e nelle sue vicinanze indica che il valore da noi ottenuto proviene dal byte immediatamente successivo alla tabella. Abbiamo dimenticato di inserire qualche valore nella tabella? Ci sono nove byte nella tabella. I valori da 0 a 9 richiedono ...dieci byte! Abbiamo dimenticato di inserire l'ultimo valore, $6F_{16}$, corrispondente alla cifra 9. Una volta aggiunto questo valore, l'esecuzione di prova funziona correttamente sia con 0 che con 9.

Le ultime due esecuzioni con i dati di prova danno

Cifra	Codice
10	6F
6B	6F

Il codice non è stato cambiato da quando abbiamo provato con il valore 9. Entrambi i valori non sono dei dati validi, per cui l'errore può trovarsi in prossimità della diramazione. A quale locazione

viene passato il controllo, dopo il salto? Aha! Direttamente all'istruzione RTS! Dobbiamo eseguire l'istruzione MOVE D1, CODE ed azzerare il risultato. La label DONE deve essere spostata nella riga precedente.

Esecuzione Definitiva

Dal momento che si tratta di un programma piuttosto semplice, può essere controllato per tutte le cifre decimali. I risultati saranno:

Cifra	Codice
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7F
7	07
8	7F
9	6F

Il risultato per il numero 6 è sbagliato; dovrebbe essere 7D. Dal momento che tutto il resto sembra essere corretto, l'errore è quasi sicuramente nella tabella. Il valore 6 della tabella è stato battuto in modo sbagliato.

Programma finale:

```
*
* CONVERSIONE DA BCD A SETTE SEGMENTI
*
* INPUT   NUMERO BCD TRA 0 E 9 NELLA LOCAZIONE DIGIT
* OUTPUT  SEQUENZA DI BIT PER UN DISPLAY A SETTE SEGMENTI
*         NELLA LOCAZIONE CODE. IL DISPLAY E` AZZERATO SE
*         LA CIFRA BCD E` MINORE DI 0 OPPURE MAGGIORE DI 9

DATA     EQU      $8000
PROGRAM  EQU      $4200

         ORG      DATA

DIGIT    DS.B     1
CODE     DS.B     1
SSEG     DC.B     $3F,$06,$5B,$4F,$66
         DC.B     $6D,$7D,$07,$7F,$6F

         ORG      PROGRAM

BCD_7SEG MOVEA.L  #SSEG,A0              PRENDI INDIRIZZO BASE DELLA TABELLA
         MOVEQ    #0,D1
         MOVE.B   DIGIT,D0              PRENDI LA CIFRA DA CONVERTIRE
         CMP.B    [illegible]D0         SE E` MAGGIORE DI 9
         BHI.S    DONE                  ALLORA VAI A DONE

         EXT.W    D0                    ALTRIMENTI ESTENDI IL SEGNO ALLA WORD
         MOVE.B   0(A0,D0,W),D1         PRENDI CODICE DALLA TABELLA

DONE     MOVE.B   D1,CODE
         RTS

         END      BCD_7SEG
```

Vi siete accorti che abbiamo anche migliorato i commenti?

Sommario degli Errori Trovati

Gli errori che abbiamo individuato in questo esempio sono quelli tipici della programmazione in linguaggio assembly. Essi comprendono:

1. **Mancata inizializzazione dei registri o delle locazioni di memoria.**
2. **Logica invertita nei salti condizionati.**
3. **Errato allineamento dei dati, nel caso di valori di un byte (anche se l'assemblatore dovrebbe segnalare che c'è qualcosa di sbagliato).**
4. **Confusione tra indirizzamento immediato e assoluto (cioè, fra dati ed indirizzi).**
5. **Dimenticare quando si verifica o meno l'estensione del segno (specialmente nel caso degli indirizzi).**
6. **Non tener conto di quali registri vengono utilizzati, ed a quale scopo, oppure battere la cifra sbagliata nell'indicare il numero di un registro.**
7. **Copiare in modo sbagliato elenchi di numeri, caratteri o istruzioni.**
8. **Trasferire il controllo ad un indirizzo sbagliato.**

19-2. DEBUGGING DI UN PROGRAMMA DI ORDINAMENTO

Questo programma esegue l'ordinamento di un elenco di numeri a 16 bit, privi di segno, in ordine decrescente. L'indirizzo iniziale dell'elenco è nella locazione di memoria LISTADDR ed il primo byte ne indica la lunghezza.

Programma iniziale: (dal diagr. di flusso della Figura 19-9)

```
00006000:                DATA     EQU     $6000
00004000:                PROGRAM  EQU     $4000
                         ;
                                  ORG     DATA

00006000:                LISTADDR DS.L    1                 IND. INIZIALE DELLA LISTA
                         ;
                                  ORG     PROGRAM

00004000: 2278 6000      BUB_SORT MOVEA.L LISTADDR,A1       PRENDI INIZIO LISTA
00004004: 7200                    MOVEQ   #0,D1
00004006: 1219                    MOVE.B  (A1)+,D1          PRENDI LUNGH. LISTA
                         ;
00004008: 534!                    SUBQ    #1,D1             N VALORI RICHIEDONO N-1 CONFRONTI
                         ;
0000400A: 45E9 0002               LEA     2(A1),A2          PRENDI IND. SECONDO ELEMENTO
0000400E: 0882 0000               BCLR    #0,D2             AZZERA FLAG DI SCAMBIO
                         ;
00004012: B549           NEXT     CMPM.W  (A1)+,(A2)+       SE (A1) <= (A2)
00004014: 6506                    BCS.S   NSWITCH           ALLORA CONTROLLA UN'ALTRA COPPIA
                         ;
00004016: 3611                    MOVE.W  (A1),D3           ALTRIMENTI SCAMBIA
00004018: 3292                    MOVE.W  (A2),(A1)         I VALORI ADIACENTI
0000401A: 3483                    MOVE.W  D3,(A2)
0000401C: 51C9 FFF4      NSWITCH  DBRA    D1,NEXT
                         ;
00004020: 0802 0000               BTST    #0,D2             C'E' STATO UNO SCAMBIO?
00004024: 66EC                    BNE     NEXT              SE SI' ALLORA DI NUOVO
                         ;
00004026: 4E75           DONE     RTS                       ALTRIMENTI DONE

                                  END     BCD_7SEG
```

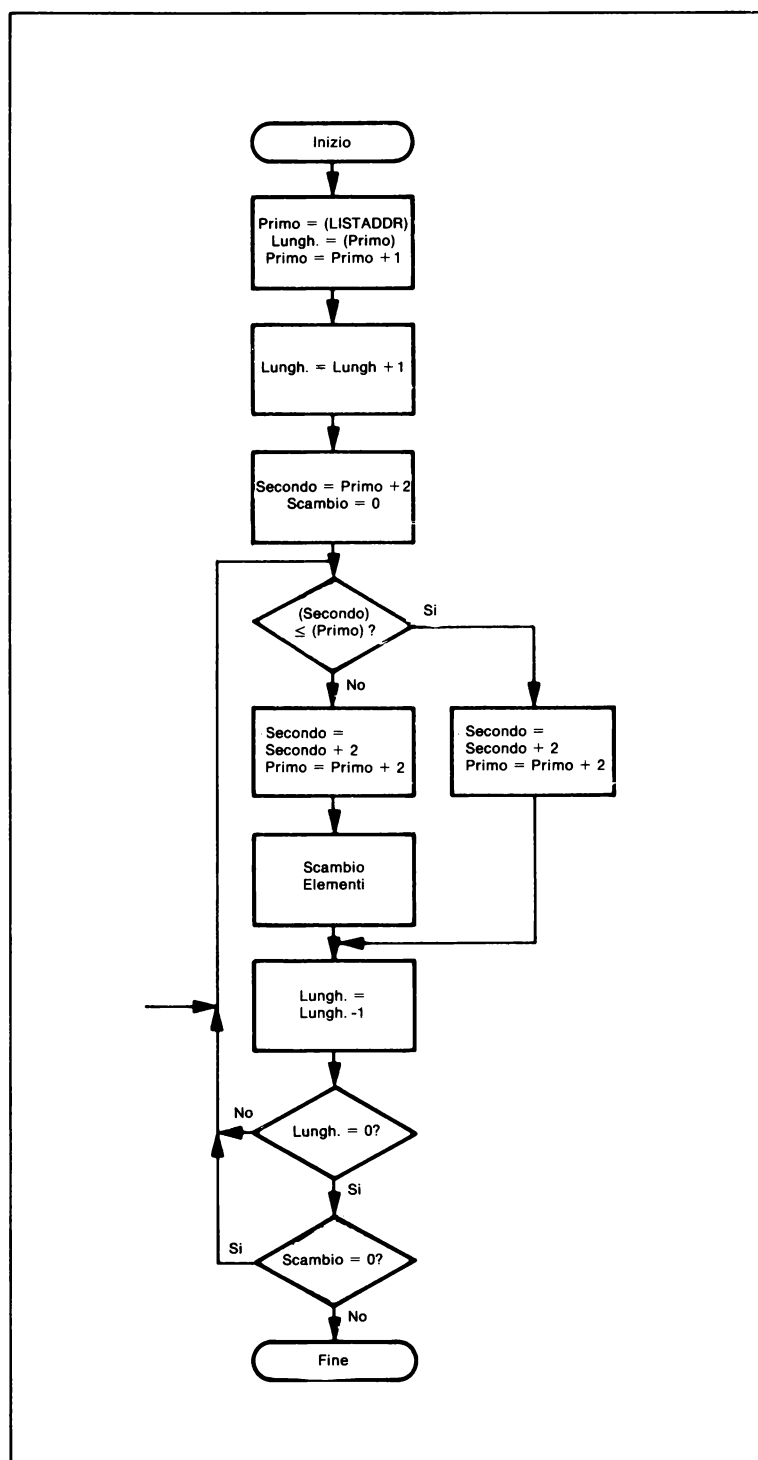



Figura 19-9. Diagramma di Flusso di un Programma di Ordinamento.

Controllo Manuale

Un primo controllo manuale ci mostra che tutti i blocchi del diagramma di flusso sono stati realizzati e che i registri utilizzati nel loop sono stati inizializzati. Dobbiamo esaminare attentamente due salti condizionati. La diramazione BCS.S NSWITCH nel loop più interno deve aver luogo se il secondo elemento è minore o uguale al primo. L'operazione che viene eseguita è (A2)-(A1). Se (A2)≠(A1), il flag di Carry sarà posto a 1, a causa del prestito. La condizione di uguaglianza (A2)=(A1) non metterà a 1 il flag di Carry, ma quello di Zero. L'istruzione BCS non gestirà correttamente un caso di uguaglianza: dobbiamo sostituirla con BLS.

Il secondo salto condizionato è BNE NEXT, che dovrebbe causare un altro passaggio all'interno del loop se si è verificato uno scambio. Il flag di scambio viene azzerato prima del loop più interno, per cui se il suo valore è 1 significa che uno scambio è avvenuto e BNE è l'istruzione adatta per una situazione di questo tipo.

L'altra cosa da controllare è il loop. Proviamo la prima iterazione manualmente. Supponiamo che la locazione di memoria LISRTADDR contenga 5000_{16}. La sezione di inizializzazione (le prime sei istruzioni) dà il seguente risultato:

```
A1 = 5001
A2 = 5003
D1 = conteggio
D2 bit# 0 = 0
```

L'effetto delle istruzioni del loop è il seguente:

```
NEXT      CMPM.W    (A1)+,(A2)+  (5001)-(5003) E AUTOINCREMENTO
          BLS.S     NSWITCH

          MOVE.W    (A1),D3      D3 := (5003)
          MOVE.W    (A2),(A1)    (5003) := (5005)
          MOVE.W    D3,(A2)      (5005) := (5003)

NSWITCH   DBRA      D1,NEXT      COUNT := COUNT - 1
```

Qui c'è qualcosa di strano. Sono stati confrontati i contenuti delle locazioni di memoria 5001 e 5003 e la locazione 5005 resta, in qualche modo, coinvolta dallo scambio. Evidentemente, abbiamo dimenticato che l'istruzione CMPM autoincrementa sia A1 che A2. Proviamo questo codice per il loop:

```
NEXT      CMPM.W    (A1)+,(A2)+  SE (A1) > = (A2)
          BLS.S     NSWITCH      CONTROLLA LA COPPIA SEG. SE
                                 C'È
```

```
         MOVE.W    -(A1),D3       ALTRIMENTI SCAMBIA I
         MOVE.W    -(A2),(A1)+    VALORI VICINI
         MOVE.W    D3,(A2)+

NSWITCH  DBRA      D1,NEXT
```

Un nuovo controllo mostrerà che questo codice è in grado di eseguire la funzione prevista.

Controlliamo, adesso, l'ultima iterazione. Supponiamo di avere tre elementi:

```
(5000) = 03
(5001) = 2015
(5003) = 1B11
(5005) = 000A
```

Ecco ciò che accade. Dopo la prima iterazione:

```
D1 = 02
A1 = 5003
A2 = 5005
```

DBRA sottrae 1 da D1 e salta a NEXT.

Dopo la seconda iterazione:

```
D1 = 1
A1 = 5005
A2 = 5007
```

Adesso A2 indica un indirizzo posto oltre l'elenco e le cose dovrebbero fermarsi a questo punto. Ma, quando DBRA sottrae 1 da D1, il risultato diventa 0 e DBRA controlla se è presente un -1. Ha luogo la diramazione ed il ciclo viene eseguito ancora una volta, o meglio una volta di troppo. Dobbiamo ovviare sottraendo 1 da D1, prima di iniziare il ciclo.

Il punto di controllo successivo nel nostro elenco sono i casi di uguaglianza. Abbiamo controllato ciò che accade con due valori uguali quando abbiamo analizzato i salti condizionati e quello è il solo caso di uguaglianza possibile in questo programma.

Controllare i Casi Banali

Cosa accade nei casi più semplici? Per prima cosa, quali sono questi casi? Quando non ci sono elementi nell'elenco? Sì, ma anche quando c'e solo un elemento nell'elenco (non ha molto senso cercare di ordinare un unico valore). Ricordate che i casi banali non sono soltanto degli elementi uguali a zero oppure l'assenza di elementi e così via. Cosa accade se abbiamo in elenco un solo valore? Il programma cercherà di ordinare 64K di memoria (se, in questa zona, si trova memoria di sola lettura, il programma continuerà all'infinito). Poche istruzioni in più per gestire queste eventualità vi

risparmieranno molti problemi ed, inoltre, esse possono essere posizionate all'esterno del loop, in modo da non prolungare eccessivamente il tempo di esecuzione. L'istruzione BLS.S DONE è la sola richiesta nel nostro programma per gestire situazioni di questo tipo. Il programma, quindi, diventerà:

```
00006000:                DATA      EQU      $6000
00004000:                PROGRAM   EQU      $4000
                         ;
                                   ORG      DATA

00006000:                LISTADDR  DS.L     1               IND. INIZIALE DELLA LISTA
                         ;
                                   ORG      PROGRAM

00004000: 2278 6000      BUB_SORT  MOVEA.L  LISTADDR,A1     PRENDI INIZIO LISTA
00004004: 7200                     MOVEQ    #0,D1
00004006: 1219                     MOVE.B   (A1)+,D1        PRENDI LUNGH. LISTA
                         ;
00004008: 5341                     SUBQ     #1,D1           N VALORI RICHIEDONO N-1 CONFRONTI
0000400A: 631E                     BLS.S    DONE            SE 0 O 1 ELEM. ALLORA DONE
                         ;
0000400C: 45E9 0002                LEA      2(A1),A2        PRENDI IND. SECONDO ELEMENTO
00004010: 0882 0000                BCLR     #0,D2           AZZERA FLAG DI SCAMBIO
00004014: 5341                     SUBQ.W   #1,D1           CORREGGI PER DBCC
                         ;
00004016: B549           NEXT      CMPM.W   (A1)+,(A2)+     SE (A1) <= (A2)
00004018: 6506                     BCS.S    NSWITCH         ALLORA CONTROLLA UN'ALTRA COPPIA
                         ;
0000401A: 3621                     MOVE.W   -(A1),D3        ALTRIMENTI SCAMBIA
0000401C: 32E2                     MOVE.W   -(A2),(A1)+     I VALORI ADIACENTI
0000401E: 34C3                     MOVE.W   D3,(A2)+
                         ;
00004020: 51C9 FFF4      NSWITCH   DBRA     D1,NEXT
                         ;
00004024: 0802 0000                BTST     #0,D2           C'E' STATO UNO SCAMBIO?
00004028: 66EC                     BNE      NEXT            SE SI' ALLORA DI NUOVO
                         ;
0000402A: 4E75           DONE      RTS                      ALTRIMENTI DONE

                                   END      BUB_SORT
```

Esecuzione di Controllo con Breakpoint

Adesso è la volta di controllare il programma su un computer o su un simulatore. Un gruppo di dati di prova potrebbe essere il seguente:

(6000) = 00005000	Indirizzo del vettore
(5000) = 02	Lunghezza del vettore
(5001) = 0100	
(5003) = 0A00	Vettore da salvare

Questa serie di dati consiste di due elementi disposti in ordine sbagliato. Il programma dovrebbe richiedere due passaggi. Il primo per scambiare gli elementi:

(5001) = 0A00
(5003) = 0100
D2 b#0 = 1 Flag di scambio

Il secondo passaggio dovrebbe soltanto trovare gli elementi già disposti in ordine decrescente e dare:

D2 bit#0 = 0 Flag di scambio

È un programma troppo lungo per un'esecuzione single-step, perciò utilizzeremo dei breakpoint. Ogni breakpoint arresterà il computer e

stamperà i contenuti dei registri chiave. Ci serviremo di quattro breakpoint e li posizioneremo nel modo seguente:

1. Dopo SUBQ.W #1,D1 per controllare l'inizializzazione.
2. Dopo CMPM.W (A1)+,(A2)+ per controllare il confronto e la diramazione.
3. Dopo MOVE.W D3,(A2)+ per controllare lo scambio.
4. Dopo BTST.B #0,D2 per controllare il completamento di un passaggio attraverso l'elenco.

Supponendo che la nostra funzione Trace ci permetta di scegliere i registri da visualizzare, selezioniamo i registri PC, D1, D2, A1, A2 ed i codici di condizione del registro di stato.

Dopo il primo breakpoint abbiamo i seguenti risultati:

```
PC = 004016
CCR = 04
D1 = 0000
D2 bit#0 = 0
```

Sono quelli previsti, per cui il programma, in questo caso, esegue l'inizializzazione in modo corretto.

Quando facciamo ripartire il nostro programma, si ha una Trap. (A questo punto, i viaggiatori meno coraggiosi del meraviglioso mondo della programmazione si abbandoneranno a scene di disperazione e grideranno "Accidenti!"). Il programma per la gestione delle Trap ci avvertirà che c'è stato un errore di indirizzo ed il codice d'istruzione, che lo ha causato, era B594 (il contatore di programma non è affidabile in un caso come questo). È il codice corrispondente all'istruzione CMPM.W (A1)+,A2+. La dimensione è di una word. A1 contiene 5001 ed A3 contiene 5003: entrambi valori dispari! La lunghezza dell'elenco è un valore di un byte e questo detrmina in A1 e A2 la presenza di valori dispari. Questo è un problema serio e la soluzione non è poi tanto semplice.

Una soluzione è quella di riscrivere il programma, in modo che legga un byte alla volta, ma si tratta di un lavoro lungo e inutile. Una seconda alternativa è di riallineare l'intero elenco, in modo che le word abbiano inizio in corrispondenza di indirizzi pari. Una terza e più semplice (per noi) alternativa è di stabilire che il primo elemento dell'elenco deve essere una word. Questo significa che dobbiamo solo cambiare il suffisso relativo alla dimensione, dopo l'istruzione MOVE. Ma fate attenzione. Questa è una modifica nelle "specifiche" e potrebbe creare dei problemi in altre parti del vostro sistema. Tuttavia, in questo caso, supponiamo che questa modifica sia possibile e cambiamo la terza istruzione del programma in:

```
MOVE.W (A1)+,D1
```

Questo errore non sarebbe stato individuato, se l'elenco avesse avuto inizio alla locazione di memoria 4FFF. Perchè?

Il nostro programma non ha nessun commento all'inizio che dica agli utenti come indicare la posizione e la lunghezza dell'elenco.

Prima di effettuare un secondo tentativo, dobbiamo cambiare l'elenco nel modo seguente:

(5000) = 0002
(5002) = 0100
(5004) = 0A00

Questa volta l'inizializzazione ci fornisce gli stessi risultati, mentre dopo il secondo breakpoint otteniamo:

PC = 004018
CCR = 00
D1 = 0000
D2 bit #0 = 0
A1 = 005004
A2 = 005006

Sono i risultati giusti e, quindi, passiamo al terzo breakpoint:

PC = 004020
CCR = 00
D1 = 0000
D2 bit # = 0
A1 = 005004
A2 = 005006

Un controllo delle locazioni di memoria indica:

(5002) = 0A00
(5004) = 0100

Proprio quello che ci attendevamo. Passiamo al quarto breakpoint:

PC = 004028
CCR = 04
D1 = FFFF
D2 bit #0 = 0
A1 = 005004
A2 = 005006

Qualcosa non va. Il bit che dovrebbe indicare che si è verificato uno scambio è ancora uguale a 0. Una rapida occhiata al loop e ci accorgiamo che nessuna istruzione cambia questo bit. La soluzione è di inserire BSET #0,D2 dopo MOVE.W D3,(A2)+.

A questo punto della procedura di debugging, la cosa più facile da fare è di mettere direttamente a 1 il bit di scambio e procedere alla seconda fase. Il breakpoint successivo è quello all'indirizzo 4016, dopo l'istruzione SUBQ.W #1,D1:

PC = 004016
SR = 00
D1 = FFFF
D2 bit #0 = 1

A1 = 005004
A2 = 005006

C'è ancora qualcosa che non va: i registri non sono reinizializzati. In questa fase dobbiamo essere certi di ritornare effettivamente all'inizio del programma per eseguire di nuovo l'inizializzazione.

Sostituiamo BNE NEXT con BNE BUB_SORT e questa volta tutto funziona nel modo giusto.

Programma Finale:

```
00006000:                DATA      EQU      $6000
00004000:                PROGRAM   EQU      $4000
                         ;
                                   ORG      DATA
00006000:                LISTADDR  DS.L     1              INDIRIZZO INIZIALE DELLA LISTA
                         ;
                                   ORG      PROGRAM
00004000: 2278 6000      BUB_SORT  MOVEA.L  LISTADDR,A1    PRENDI INIZIO LISTA
00004004: 7200                     MOVEQ    #0,D1
00004006: 3219                     MOVE.W   (A1)+,D1       PRENDI LUNGH. LISTA
                         ;
00004008: 5341                     SUBQ     #1,D1          N VALORI RICHIEDONO N-1 CONFRONTI
0000400A: 6322                     BLS.S    DONE           SE <= 0 ALLORA DONE
                         ;
0000400C: 45E9 0002                LEA      2(A1),A2       PRENDI IND. SECONDO ELEMENTO
00004010: 0882 0000                BCLR     #0,D2          AZZERA FLAG DI SCAMBIO
00004014: 5341                     SUBQ.W   #1,D1          CORREGGI PER DBCC
                         ;
00004016: B549           NEXT      CMPM.W   (A1)+,(A2)+    SE (A1) <= (A2)
00004018: 630A                     BLS.S    NSWITCH        ALLORA CONTROLLA UN'ALTRA COPPIA
                         ;
0000401A: 3621                     MOVE.W   -(A1),D3       ALTRIMENTI SCAMBIA
0000401C: 32E2                     MOVE.W   -(A2),(A1)+    I VALORI ADIACENTI
0000401E: 34C3                     MOVE.W   D3,(A2)+
                         ;
00004020: 08C2 0000                BSET     #0,D2          AZZERA FLAG DI SCAMBIO
                         ;
00004024: 51C9 FFF0      NSWITCH   DBRA     D1,NEXT
                         ;
00004028: 0802 0000                BTST     #0,D2          C'E' STATO UNO SCAMBIO?
0000402C: 66D2                     BNE      BUB_SORT       SE SI' ALLORA DI NUOVO
                         ;
0000402E: 4E75           DONE      RTS                     ALTRIMENTI DONE

                                   END      BUB_SORT
```

Questo programma ha ancora bisogno di alcuni commenti nella parte iniziale.

Altre Esecuzioni di Prova

Chiaramente, non possiamo provare tutti i casi possibili. Ecco ancora qualche tentativo che possiamo fare a scopo di debugging:

1. Nessun valore nell'elenco:

(6000) = 00005000
(5000) = 0000

2. Un solo elemento nell'elenco:

(6000) = 00005000
(5000) = 0001

3. Valori casuali con due elementi uguali:

(6000) = 00008200
(8200) = 0004 Numero di elementi in elenco

```
(8202) = 8345
(8204) = 0001  Vettore da salvare
(8206) = 0001
(8208) = 4657
```

Sommario degli Errori Trovati

Con questo programma, abbiamo acquistato familiarità con altri errori, che certamente vi capiterà di incontrare nella vostra carriera di programmatori dell'MC68000. Sono i seguenti:

1. **Indicare la condizione sbagliata nei salti condizionati** (ancora una volta, ma si tratta di un errore piuttosto frequente).
2. **Dimenticare gli effetti dell'autoincremento /autodecremento o i valori dei puntatori.**
3. **Dimenticare che DBcc controlla la presenza di un -1 o calcolare in modo sbagliato la lunghezza di un vettore** (lunghezza = fine - inizio +1).
4. **Gestire in modo errato alcuni casi banali o dei casi di uguaglianza o, magari, trascurare del tutto alcuni dei casi banali.**
5. **Cercare di indirizzare word e long word a indirizzi dispari.** Questo accade facilmente con strutture dati definite molto approssimativamente, che richiedono sia istruzioni di un byte che di una word.
6. **Dimenticare di mettere a 1 o azzerare dei flag**
7. **Dimenticare di reinizializzare i loop più interni delle strutture nidificate.**

BIBLIOGRAFIA

1. Per maggiori informazioni sugli analizzatori logici, vedi:

   Brock, G. "Logic-State Analyzers Seek Out Microprocessor-System Faults," *EDN* January 5, 1980, pp. 137-40.
   Lorentzen, R. "Logic Analyzers Finish What Development System Start," *Electronic Design,* March 29, 1980, pp. 81-85.
   Marshall, J. "Digital Analysis Instruments," *EDN,* January 20, 1980, pp. 141-43. Ogdin, C.A. "Setting up a Microcomputer Design Laboratory," *Mini-Micro Systems,* May 1979, pp. 87-94.
   Spector, I.H., and Muething, R. "Logic Analyzer Deploys Its Full Strength," *Electronic Design,* March 29, 1980, pp. 177-214.
2. Weller, W.J. *Assembly Level Programming for Small Computers.* Lexington, Mass.: Lexington Books, 1975, Chapter 23.
3. Baldridge R.L. "Interrupts Add Power, Complexity to Microcomputer System Design," *EDN,* August 5, 1977, pp. 67-73.

CAPITOLO 20

COLLAUDO

Il collaudo di un programma[1] è strettamente correlato alla fase di debugging. Infatti, utilizzeremo in entrambi i casi gli stessi dati; ad esempio,

- **Casi banali,** come in assenza di dati o nel caso di un unico dato.
- **Casi speciali,** che, per un qualsiasi motivo, il programma gestisce in modo anomalo.
- **Casi semplici,** che servono a provare determinate parti del programma.

Nel caso del programma di conversione di una cifra decimale in un codice a sette segmenti, presentato nel Capitolo 19, questo elenco racchiude tutte le situazioni possibili. I dati di prova saranno:

- I numeri da 0 a 9
- Il caso limite 10
- Il valore casuale $6B_{16}$

Il programma non prevede nessun altro caso, per cui, questa volta, **il debugging ed il collaudo sono praticamente la stessa cosa.**

Per il programma di ordinamento il problema è più complesso. Il numero degli elementi varia da 0 a 255 ed ogni elemento può assumere uno qualsiasi dei valori compresi in questo stesso intervallo. Il numero dei casi possibili è enorme ed anche il programma è abbastanza complesso. Con quale criterio possiamo scegliere dei dati di prova che ci diano una completa certezza riguardo all'affidabilità del programma? **In questo caso il collaudo richiede alcune scelte in fase di progettazione.** Il problema del collaudo diventa particolarmente complesso, soprattutto se un programma dipende da sequenze di dati in tempo reale. Come selezionare i dati, generarli e presentarli ad un microcomputer in modo sufficientemente realistico?

STRUMENTI PER IL COLLAUDO

La maggior parte degli strumenti indicati in precedenza per il debugging si riveleranno utili anche in fase di collaudo. Gli analizzatori logici possono servire per il controllo dell'hardware, i simulatori[2] per

quello del software. Sono impiegati anche altri strumenti ed altre tecniche, fra cui:

1. **Simulazioni di I/O,** in grado di simulare una serie di periferiche, mediante un solo dispositivo di input ed uno di output.
2. **Emulatori in circuito,** che consentono di collegare il prototipo ad un sistema di sviluppo o ad un pannello di controllo e di provarlo[3].
3. **Simulatori di ROM,** che possono essere cambiati come la RAM, ma, per il resto, si comportano come le ROM o le PROM, che verranno utilizzate nel sistema finale.
4. **Sistemi operativi in tempo reale,** che forniscono input o interrupt in determinati momenti (oppure casualmente) e segnalano la comparsa di eventuali output. Consentono anche di inserire dei breakpoint e di poter disporre di una funzione Trace, sempre in tempo reale.
5. **Emulazioni** (spesso su computer microprogrammabili), che consentono esecuzioni in tempo reale ed I/O programmabili.
6. **Interfacce,** che permettono ad un altro elaboratore di controllare il sistema di I/O e di collaudare il programma di un microcomputer
7. **Programmi di collaudo,** che controllano ogni diramazione all'interno di un programma, per individuare eventuali errori logici.
8. **Programmi per la generazione di dati,** che possono essere casuali o scelti secondo criteri ben precisi.

Alterazioni delle condizioni di svolgimento delle prove

Esistono dei teoremi formali per il collaudo, ma, in pratica, sono utilizzabili solo nella verifica di brevi programmi. È necessario che l'apparecchiatura di collaudo non invalidi la prova, alterando le condizioni in cui questa si svolge. Spesso certe apparecchiature condizionano i segnali in ingresso e in uscita con la presenza di buffer, di latch, ecc. Se questi non saranno disponibili anche nel sistema reale, finiremo per registrare un comportamento del tutto diverso.

Inoltre, il software supplementare, impiegato durante il collaudo, può utilizzare dello spazio di memoria o parte del sistema di interrupt, oltre a consentire, in certi casi, di ripristinare il sistema dopo un errore ed a fornire tutta una serie di altre caratteristiche che non si ritroveranno nel sistema finale. Le condizioni in cui avviene il collaudo del software devono essere realistiche al pari di quelle necessarie al collaudo dell'hardware, poichè un eventuale problema a livello di software risulterà altrettanto critico di uno dovuto all'hardware.

Imprecisione delle emulazioni e delle simulazioni

Le emulazioni e le simulazioni, naturalmente, non sono mai precise. Generalmente, sono sufficienti per il controllo della logica, ma non servono molto nel collaudo delle interfacce o della temporizzazione. D'altra parte, le apparecchiature di collaudo in tempo reale forniscono soltanto un quadro generale della logica del programma e possono alterare l'interfacciamento e la temporizzazione.

Numero dei campioni necessari

Sono pochi i programmi reali che possono essere collaudati per tutti i casi possibili. Al momento del collaudo, il progettista deve limitarsi a scegliere un campione di dati sufficientemente rappresentativo.

Collaudo Strutturato

Naturalmente, il collaudo dovrebbe far parte della procedura di sviluppo complessiva. **La progettazione top-down e la programmazione strutturata tengono presenti le necessità del collaudo, già durante la fase di progettazione. È il cosiddetto collaudo strutturato.** Ciascuno dei moduli di un programma strutturato viene controllato separatamente. **Il collaudo, al pari della progettazione, dovrà essere modulare, strutturato e del tipo top-down.**

Casi Speciali

Ma resta ancora il problema di selezionare i dati di prova per un determinato modulo. Per prima cosa, il progettista deve elencare tutti i casi speciali riconosciuti da un programma:

- Casi banali
- Casi di uguaglianza
- Situazioni speciali

I dati di prova dovrebbero includere tutte queste eventualità.

Formare delle Classi di Dati

Successivamente bisogna identificare ciascuna classe di dati che gli statement del programma sono in grado di distinguere. Possono essere:

- Numeri positivi o negativi
- Numeri al di sopra o al di sotto di un particolare valore soglia
- Dati che includono o meno sequenze o caratteri particolari
- Dati che sono presenti o meno in un momento particolare

Attenzione: **una scelta che prevede due possibilità raddoppia il numero delle classi,** in quanto è necessario collaudare entrambe le diramazioni possibili. Perciò, tre salti condizionati daranno 2 X 2 X 2 = 8 classi. **Un altro motivo per utilizzare sempre dei moduli brevi e generici è quello di limitare i dati campione necessari.**

Selezionare Dati da una Classe

Adesso bisogna separare le classi, a seconda che il programma fornisca un risultato diverso per ciascun valore di quella classe (come in una tabella) oppure dia sempre lo stesso risultato (ad es. per avvertire che un parametro è al di sopra di un certo valore soglia). In entrambi i casi, si possono utilizzare tutti gli elementi, se il loro numero complessivo non è troppo grande, oppure un campione che comprenda tutti i casi limite ed almeno un valore scelto a caso. Tabelle di numeri casuali si trovano in molti testi e, inoltre, generatori di numeri casuali sono disponibili su gran parte dei microcomputer.

Bisogna fare attenzione alle distinzioni che non risultano abbastanza evidenti. Ad esempio, il microprocessore MC68000 considera negativo un numero ad 8 bit privo di segno e maggiore di 127: è un aspetto da tener presente con istruzioni di salto che dipendano dal flag di Negativo (Segno).

ESEMPI

20-1. COLLAUDARE UN PROGRAMMA DI ORDINAMENTO

I casi speciali sono piuttosto evidenti:

- **Assenza di elementi nel vettore**
- **Un solo elemento,** la cui grandezza può essere scelta a caso

L'altro caso speciale da tener presente è quello in cui gli elementi sono uguali.

Ci può essere qualche problema con i segni e la lunghezza dei dati. Si ricordi che il vettore non deve contenere più di 255 elementi.

Possiamo controllare se il segno del byte che indica il numero degli elementi non ha nessun effetto, eseguendo metà delle prove con valori compresi fra 128 e 255 e l'altra metà con valori fra 2 e 127. La grandezza degli elementi dovrebbe essere scelta a caso, per evitare una propensione inconscia per numeri piccoli, cifre decimali (anzichè esadecimali) o sequenze regolari.

20-2. COLLAUDARE UN PROGRAMMA ARITMETICO

Si presuppone, in questo caso, che un precedente controllo abbia accertato che il numero sia della lunghezza prevista e sia formato da cifre valide. Dal momento che il programma non effettua altre distinzioni, i dati di prova dovrebbero essere scelti a caso. L'ideale sarebbe poter disporre di una tabella o di un generatore di numeri casuali compresi fra 0 e 255.

REGOLE PER IL COLLAUDO

Una progettazione accurata semplifica la fase di collaudo. Ecco alcune regole da seguire:

1. **Eliminare prima i casi banali, senza introdurre distinzioni superflue.**
2. **Evitare i casi speciali,** dato che essi aumentano il tempo necessario al debugging ed al collaudo.
3. **Controllare la validità dei dati,** prima che questi siano elaborati.
4. **Evitare distinzioni non volute,** soprattutto nella gestione dei numeri provvisti di segno o nell'impiego di istruzioni destinate a questo tipo di gestione.
5. **Controllare manualmente i casi limite.** Assicuratevi di aver definito con esattezza ciò che deve accadere in queste situazioni.
6. **Rendere il programma sufficientemente generico.** Ogni distinzione e routine separata richiedono ulterori collaudi.
7. **Usare la progettazione top-down e la programmazione modulare allo scopo di rendere modulare anche il collaudo.**

CONCLUSIONI

Le fasi di debugging e di collaudo sono le figliastre del processo di sviluppo del software. La maggioranza dei progetti riservano loro troppo poco tempo e la gran parte dei manuali le trascura completamente. Ma i progettisti ed i manager le ritengono spesso le fasi più costose e lunghe. È difficile valutare i risultati ottenuti e, a volte, perfino ottenere dei risultati. Il collaudo ed il debugging del software destinato ai microprocessori sono ancor più complessi, in quanto i potenti strumenti hardware o software che possono essere usati nei computer più grandi, raramente sono disponibili su dei microcomputer.

Un progettista deve pianificare accuratamente le fasi di debugging e di collaudo. Ecco alcune indicazioni in proposito:

Pianificazione del debugging e del testing

1. **Cercate di scrivere programmi facili da verificare e collaudare.** La programmazione modulare, quella strutturata e la progettazione top-down sono tecniche molto utili in tal senso.
2. **Pianificare le fasi di debugging e collaudo, già durante la definizione del problema.** Stabilite, fin dall'inizio, quali dati dovranno essere generati e quale apparecchiatura sarà necessaria.
3. **Eseguire il debugging ed il collaudo di ciascun modulo, facendo uso della progettazione top-down.**
4. **Eseguire sistematicamente il debugging della logica di ciascun modulo.** Usate liste di controllo, breakpoint ed esecuzioni single-step. Se la logica del programma è piuttosto complessa, considerate la possibilità di utilizzare un simulatore software.

5. **Controllate sistematicamente la temporizzazione di ciascun modulo, se questo può essere un elemento critico.** Se utilizzato in modo appropriato, un oscilloscopio è in grado di risolvere molti problemi. Nel caso di una temporizzazione particolarmente complessa, servitevi di un analizzatore logico.
6. **Assicuratevi che i dati campione siano sufficientemente rappresentativi.** Tenete presenti le classi di dati che il programma è capace di riconoscere, includendo tutti i casi banali e quelli speciali.
7. **Se il programma gestisce ogni elemento in modo diverso oppure il numero dei casi è eccessivo, scegliete i dati di prova in modo casuale.**
8. **Documentate tutte le prove eseguite.** Se in seguito troverete degli errori, non dovrete ripetere prove già effettuate.

BIBILIOGRAFIA

1. G.J. Myers. *The Art of Software Testing, Wiley,* New York, 1979.
   R.C. Tausworthe. *Standardized Development of Computer Software,* Prentice-Hall, Englewood Cliffs, N.J., Vol. 1, 1977, Chapter 9; Vol. 2, 10979, Chapters 14 and 15.
   E. Yourdon. *Techniques of Program Structure and Design,* Prentice-Hall, Englewood Cliffs, N.J., 1975, Chapter 7.
2. F.J. Langley. "Simulating Modular Microcomputers," *Simulation,* May 1979, pp. 141-54.
   L.A. Leventhal. "Design Tools for Multiprocessor Systems," *Digital Design,* October 1979, pp. 24-26.
   F.I. Parke et al. "An Introduction to the N.mPc Design Environment," *Proceedings of the 1979 Design Automation Conference,* San Diego, Ca., pp. 513-19.
3. R. Francis and R. Teitzel. "Realtime Analyzer Aids Hardware/Software Integration," *Computer Design,* January 1980, pp. 140-50.
4. H.R. Burris. "Time-Scaled Emulations of the 8080 Microprocessor, "*Proceedings of the 1977 National Computer Conference,* pp. 937-46.
5. R.A. DeMilo et al. "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* April, 1978, pp. 34-41.
   W.F. Dalton. "Design Microcomputer Software," *Electronics,* January 19, 1978, pp. 97-101.
6. R.D. Grappel and J. Hemenway. "EDN Software Tutorial: Pseudorandom Generators," *EDN,* May 20, 1980, pp. 119-23.
   T.G. Lewis. *Distribution Sampling for Computer Simulation,* Lewington Books, Lexington, Mass., 1975.
   R.A. Mueller et al. "A Random Number Generator for Microprocessors," *Simulation, April 1977, pp. 123-27.*

CAPITOLO 21

MANUTENZIONE E RIPROGETTAZIONE

Fattori che rendono necessario modificare un programma

La manutenzione di un programma comporta sempre un certo grado di riprogettazione. Al momento del suo impiego, può accadere che un programma non funzioni nel modo previsto, a causa di un problema che non è stato individuato durante il debugging ed il collaudo. Altre volte, un programma funziona in modo corretto, ma del tutto inefficiente (con un tempo di risposta molto lungo o rendendo necessaria tutta una complessa serie di operazioni da parte dell'utente). In certi casi, un produttore decide di modificare un sistema di controllo, in modo che sia utilizzabile anche su configurazioni di tipo diverso. Inevitabilmente, accade che qualcuno trovi un modo di utilizzare un microcomputer che al progettista non era mai passato per la mente: le necessità di un utente cambiano spesso in modo del tutto imprevedibile. **Può accadere, quindi, che sia necessario modificare un programma o un sistema, anche se questo funziona correttamente.**

Casi in cui la riprogettazione è utile

Alcune volte, un progettista deve aumentare, anche di un solo microsecondo, la velocità di esecuzione di un programma o ridurre, anche di un solo byte, la memoria occupata. Dal momento che hanno cominciato ad essere disponibili memorie sempre più grandi su singoli chip, il problema della memoria non è poi così grave. Il fattore tempo, naturalmente, è critico solo in alcuni tipi di applicazioni. Nella maggior parte dei casi, invece, un microprocessore passa gran parte del suo tempo ad aspettare un segnale proveniente da dispositivi esterni e, perciò, la velocità del programma non rappresenta un elemento fondamentale.

COSTI DI RIPROGETTAZIONE

Migliorare anche di poco le prestazioni di un programma è raramente così importante come alcuni programmatori amano far credere. Innanzitutto, questo **comporta dei costi molto alti, per i seguenti motivi:**

1. Richiede altro tempo al programmatore, e questo è una delle voci che incidono maggiormente sul costo del software.

2. Va a detrimento della strutturazione e della semplicità, rendendo più complesse le fasi di collaudo e debugging.
3. Il programma richiede un'ulteriore documentazione.
4. Il programma diventa difficile da ampliare, mantenere o riutilizzare.

In secondo luogo, spesso un costo unitario inferiore e delle prestazioni migliori non sono veramente importanti. Un costo più basso e prestazioni migliori garantiranno delle vendite superiori? Oppure si otterranno risultati migliori, facilitando ancor più l'utilizzazione da parte dell'utente? **Le sole applicazioni che sembrano giustificare questo sforzo sono quelle di grande volume, basso costo e basse prestazioni, dove il prezzo di un altro chip di memoria va ben al di là del costo richiesto da un ampliamento del software.** Negli altri casi, vi accorgerete che si tratta di un gioco inutile e molto costoso.

RIORGANIZZAZIONI DI MAGGIORE O MINORE IMPORTANZA

Grado delle modifiche richieste

Tuttavia, dovendo riprogettare un programma, i suggerimenti che seguono vi saranno di valido aiuto. Per prima cosa, bisogna stabilire con esattezza di quanto devono essere migliorate le prestazioni o di quanto deve essere ridotta l'area di memoria occupata. Se viene richiesto un miglioramento che non va oltre il 25%, sarà possibile ottenerlo mediante una semplice riorganizzazione del programma. Se, invece, si tratta di un miglioramento superiore al 25%, significa che c'è stato un errore fondamentale in fase di progettazione e saranno necessarie drastiche modifiche sia all'hardware che al software. Analizzeremo prima i problemi connessi ad una semplice riorganizzazione e, successivamente, quelli legati a delle modifiche sostanziali. Ridurre la memoria occupata è molto importante se il programma è destinato ad un microcomputer che dispone di uno o due chip di memoria soltanto. L'impiego di questo tipo di microcomputer, in grado di funzionare in modo autonomo, riduce notevolmente i costi delle applicazioni di portata limitata.

RISPARMIO DI MEMORIA

Le procedure seguenti servono a ridurre la memoria occupata dai programmi in linguaggio assembly;

Quando ridurre la memoria occupata

1. **Sostituire delle sequenze, che si ripetono spesso, con delle subroutine.** Assicuratevi, tuttavia, che le istruzioni JSR (Jump to Subroutine) e RTS (Return from Subroutine) non finiscano per eliminare gran parte del guadagno ottenuto. Di solito, questa sostituzione rende i programmi piu lenti a causa dei vari trasferimenti di controllo.

2. **Mettere i dati nei registri, tutte le volte che questo è possibile.** I puntatori nei registri utilizzano meno memoria degli indirizzi diretti e indicizzati, a meno che la necessaria inizializzazione non finisca per eliminare il guadagno ottenuto.
3. **Utilizzare quando è possibile lo stack o l'indirizzamento con autodecremento.** Il puntatore dello stack o gli indirizzi sono aggiornati automaticamente, una volta utilizzati, per cui non sono necessarie esplicite istruzioni di aggiornamento.
4. **Eliminare le istruzioni di salto.** Cercate di riorganizzare il programma in altro modo.
5. **Usare i risultati intermedi di precedenti sezioni del programma.**
6. **Utilizzare istruzioni di shift per operare su bit posti alle estremità di un byte, di una word o di una long word.**
7. **Impiegare le forme abbreviate delle istruzioni, come ADDQ, MOVEQ e così via.**
8. **Usare diramazioni relative, anzichè salti con indirizzamento diretto.**
9. **Quando dovete usare l'indirizzamento assoluto diretto, impiegate la versione breve, invece di quella lunga.**
10. **Utilizzare degli algoritmi, anzichè delle tabelle, per il calcolo di espressioni aritmetiche o logiche o per effettuare la conversione di un codice.** Ricordate che questa sostituzione può rallentare il programma.
11. **Ridurre la dimensione di tabelle matematiche, mediante interpolazioni fra i valori presenti.** Anche in questo caso si risparmia memoria a scapito del tempo di esecuzione.
12. **Sfruttare l'istruzione LEA** per eseguire operazioni aritmetiche e calcolare indirizzi indiretti, indicizzati e relativi, che, in seguito, saranno utilizzati ripetutamente.
13. **Usare l'indirizzamento indicizzato, anzichè quello assoluto, per la gestione dei PIA** ed in altre situazioni in cui si fa riferimento **a parecchi indirizzi vicini l'uno all'altro.**
14. **Cercate di sostituire sequenze di istruzioni di diramazione con diramazioni singole.** Questo si ottiene modificando il tipo di elaborazione o utilizzando salti condizionati che dipendano dal valore di determinati flag. Esaminare attentamente gli effetti di diramazioni del tipo di BGE, BGT, BHI, BLE, BLS e BLT; possono rivelarsi utili in situazioni che differiscono notevolmente da quelle suggerite dai relativi mnemonici.
15. **Usare istruzioni come BTST e CMP, che influiscono sui flag, senza modificare i registri o le locazioni di memoria,** in modo da conservare i dati e riutilizzarli successivamente.
16. **Analizzare tabelle complesse per verificare che non contengano un numero eccessivo di informazioni.** Tabelle con dati superflui sono spesso le maggiori responsabili di un uso eccessivo di memoria.
17. **Due parti separate di un programma possono utilizzare un'unica area dati,** come un buffer di I/O, soprattutto quando nessuna delle due dipende dai dati dell'altra.

RIDURRE IL TEMPO DI ESECUZIONE

Sebbene alcuni dei metodi impiegati per limitare l'occupazione della memoria permettano anche di ridurre il tempo di esecuzione, i risultati migliori in questo senso li otterremo concentrandoci sui loop che vengono eseguiti più spesso. Infatti, eliminando completamente un'istruzione che viene eseguita soltanto una volta, si risparmiano al massimo pochi microsecondi. Ma un risparmio ottenuto in un loop eseguito ripetutamente va moltiplicato per parecchie volte.

Influenza dei loop sul tempo di esecuzione

Perciò, **per ridurre il tempo di esecuzione, bisogna procedere in questo modo:**

1. **Stabilire quante volte un determinato ciclo di programma viene ripetuto.** Questo si può fare manualmente o con l'aiuto di un simulatore software o altri metodi simili.
2. **Esaminare i loop nell'ordine determinato dalla frequenza di esecuzione,** a partire da quello ripetuto più volte e continuando finchè non si sono ottenuti i risultati desiderati.
3. **Per prima cosa, controllate se c'è qualche operazione che può essere portata fuori da un loop,** come calcoli ripetitivi, dati che possono essere memorizzati in un registro o sullo stack, dati o indirizzi che possono essere messi nella pagina ad indirizzamento diretto, casi speciali o errori che possono essere gestiti altrove, ecc. Questo comporta spesso una fase di inizializzazione più lunga e richiede più memoria, ma consente di risparmiare tempo.
4. **Cercare di eliminare le istruzioni Jump,** che richiedono molto tempo. Alcune volte, è possibile farlo, cambiando le condizioni iniziali, soprattutto se le modifiche consentono di eseguire i test alla fine di un ciclo, anzichè all'inizio.
5. **Eliminare le subroutine,** anche dovendo ripetere più volte sequenze di istruzioni identiche. Questo servirà a risparmiare almeno un'istruzione BSR e una RTS.
6. **Usare lo stack per memorizzare temporaneamente dei dati,** soprattutto quando può esservi utile l'ordinamento che esso fornisce automaticamente.
7. **Utilizzare i metodi indicati per risparmiare memoria e che, allo stesso tempo, riducono il tempo di esecuzione.**
8. **Non prendere assolutamente in considerazione le istruzioni che vengono eseguite una volta sola.** Qualunque modifica di questo tipo non garantisce nessun guadagno apprezzabile e aumenta, soltanto, la probabilità di nuovi errori.
9. **Evitare, se possibile, gli indirizzamenti indicizzati e indiretti,** dato che richiedono un tempo più lungo.
10. **Utilizzare tabelle, invece di algoritmi;** ricorrete alle tabelle tutte le volte che è possibile, anche se dovrete ripetere più volte gli stessi valori all'interno di una tabella.

RIORGANIZZAZIONI SOSTANZIALI

Se dovete ottenere un incremento della velocità di esecuzione superiore al 25% o una diminuzione della memoria occupata di uguale misura, non potete limitarvi a riorganizzare il codice. Le probabilità di ottenere un tale risultato sono veramente scarse, a meno di non ricorrere ad un esperto. **Un cambiamento sostanziale darà, senz'altro, risultati migliori.**

ALGORITMI MIGLIORI

Metodi per ridurre le difficoltà delle modifiche

La modifica più ovvia è un algoritmo migliore. Soprattutto nel caso di ordinamenti, ricerche o calcoli matematici, è sempre possibile trovare un metodo più breve o più veloce in uno dei vari manuali. Indicazioni sui possibili algoritmi si trovano spesso su molte riviste o presso gruppi di professionisti. Consultate la bibliografia alla fine di questo capitolo per alcune delle fonti più importanti.

ALTRE MODIFICHE IMPORTANTI

L'hardware può sostituire il software. Contatori, registri di shift, unità aritmetiche, moltiplicatori hardware ed altri dispositivi veloci consentono un risparmio di tempo e di memroria. Apparecchiature di calcolo, UART, tastiere, encoder ed altri dispositivi più lenti servono a ridurre l'uso della memoria, anche se sono lenti. Interfacce seriali e parallele compatibili, progettate appositamente per essere usate con il 6809 o il 6502, garantiscono un certo risparmio di tempo, riducendo il sovraccarico della CPU.

Modifiche altrettanto utili sono:

1. **Una CPU con una word più lunga risulterà più veloce** con dati di parecchi byte. Una CPU di questo tipo riduce la memoria occupata. I processori a 16 bit, ad esempio, utilizzano la memoria in modo più efficiente di quelli ad 8 bit, in quanto molte delle loro istruzioni sono lunghe una word.
2. **In certi casi, esistono altre versioni di una stessa CPU che funzionano a frequenze di clock più elevate.** Ricordate, però, che avrete bisogno anche di memorie e porte di I/O più veloci e sarà necessario ricalcolare eventuali routine di ritardo.
3. **Due CPU possono eseguire il lavoro in parallelo** o separatamente, qualora sia possibile suddividere i compiti e risolvere il problema della comunicazione.
4. **Un processore microprogrammato appositamente può eseguire lo stesso programma con maggiore rapidità.** Il costo, tuttavia, sarà molto più alto, anche utilizzando un'emulazione fuori banco.

5. **Ricorrere a dei compromessi fra tempo e memoria.** Tabelle di riferimento e funzioni di ROM saranno più veloci degli algoritmi, ma occuperanno più memoria.

Decidere una Modifica Sostanziale

La necessità di un notevole miglioramento, in genere, è dovuta alla mancanza di una pianificazione adeguata nelle fasi di definizione e progettazione. Al momento della definizione del problema dovete stabilire quale processore e quali metodi adottare. Una valutazione errata comporterà dei costi molto alti. Una soluzione a basso costo può comportare un tempo di sviluppo molto più lungo e costoso. Non fate le cose in modo superficiale; la soluzione migliore è quella di eseguire il progetto in modo accurato e di far tesoro degli eventuali insuccessi. **Se avete adottato metodi come i diagrammi di flusso, la programmazione modulare, quella strutturata, la progettazione top-down e una documentazione adeguata, anche una modifica di una certa importanza non sarà un compito eccessivamente arduo.**

BIBLIOGRAFIA

Carnahan, B., et al. *Applied Numerical Methods,* Wiley, New York, 1969.

Chen, T.C. "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots," *IBM Journal of Research, and Development,* Volume 18, pp. 380-388, July 1972.

Collected Algorithms from ACM, ACM Inc, P.O. Box 12105, Church Street Station, New York, 10249.

Despain, A.M. "Fourier Transform Computers Using CORDIC Iterations," *IEEE Transactions on Computers,* october 1974, pp. 993-1001.

Edgar, A.D. and S.C. Lee. "FOCUS Microcomputer Number System," *Communications of the ACM,* March 1979, pp. 166-177.

Hwang, K. *Computer Arithmetic, Wiley, New York, 1978.*

Knuth, D.E. The Art of Computer Programming Volume 1: Fundamental Algorithms; The Art of Computer Programming Volume 2: Seminumerical Algorithms; The Art of Computer Programming, Volume 3 Sorting and Searching, Addison-Wesley, Reading, Mass. 1967-1969.

Luke, Y.L. *Algorithms for the Computation of Mathematical Functions,* Academic Press, New York, 1977.

Schmid, H. *Decimal Computation,* Wiley-Interscience, New York, 1974.

Nuovi metodi per la realizzazione di operazioni aritmetiche sugli elaboratori sono spesso trattati al Symposium on Computer Arithmetic, che si tiene ogni tre anni. I relativi documenti (a partire dal 1969) sono disponibili presso la IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamos, Calif. 90720.

SEZIONE V

Il Set di Istruzioni dell'MC68000

Il Capitolo 22 e le appendici che lo seguono contengono una descrizione del set di istruzioni dell'MC68000. Il Capitolo 22 esamina nei dettagli ciascuna istruzione; le appendici forniscono un sommario di queste informazioni.

CAPITOLO 22

DESCRIZIONE DELLE SINGOLE ISTRUZIONI

In questo capitolo presentiamo le istruzioni dell'MC68000 in ordine alfabetico, fornendo per ciascuna di esse una descrizione dettagliata. Le informazioni contenute in questo capitolo le potrete ritrovare, in forma più sintetica, nelle Appendici A, B e C. La descrizione di ciascuna istruzione viene accompagnata da un diagramma, che ne illustra l'esecuzione. Dal momento che l'MC68000 ha moltissimi tipi di indirizzamento, è stato praticamente impossibile, per motivi di spazio, fornire una descrizione dettagliata di tutti quelli utilizzabili con ogni singola istruzione e ci limiteremo, pertanto, alla loro elencazione.

ABC (Addizione Decimale con Extend tra Valori nei Registri)

Questa istruzione somma il contenuto del registro dati sorgente ed il valore del flag di Extend (X) al contenuto del registro dati destinazione. Il risultato viene depositato nel registro dati destinazione. L'addizione viene eseguita utilizzando l'aritmetica BCD (decimale codificata binaria). Sono interessati solo gli otto bit meno significativi dei registri dati.

Il codice oggetto dell'istruzione è:

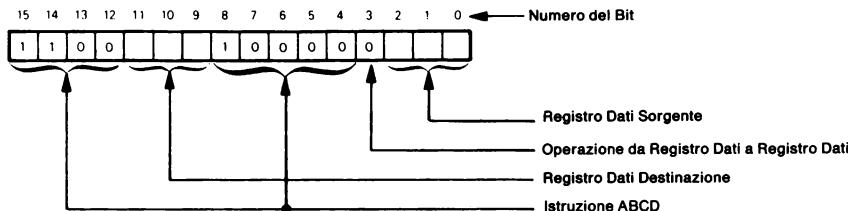
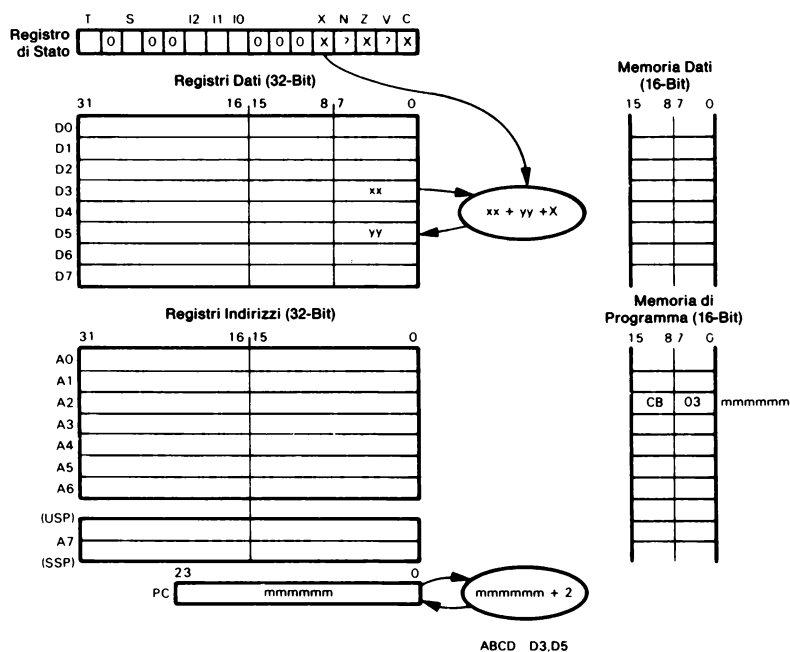
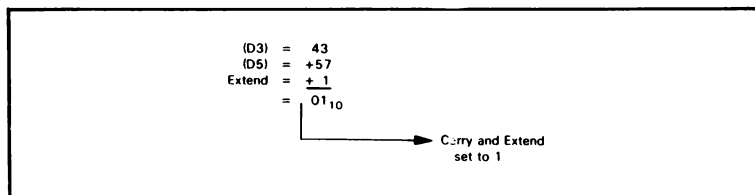



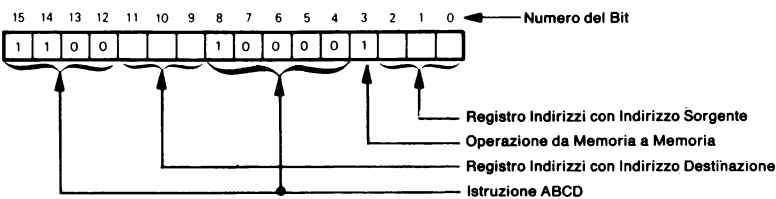
Figura 22-1. Esecuzione dell'Istruzione ABCD con gli Operandi nei Registri Dati.

La figura 22-1 mostra l'esecuzione dell'istruzione ABCD con il registro D3 come registro sorgente e D5 come registro destinazione. Supponiamo che xx = 43_{10}, Extend = 1 e yy = 57_{10}. Dopo che il processore ha eseguito l'istruzione ABCD D3,D5. il contenuto di D5 sarà 01_{10}.

I bit dall'8 al 31 dei registri dati non sono interessati da questa istruzione.

I flag di Carry (C) e di Extend (X) sono posti a 1, se, in seguito all'operazione, si verifica un riporto decimale; altrimenti sono azzerati. Il flag di Zero (Z) viene azzerato se il risultato è diverso da zero; resta immutato se il risultato è zero. I flag N e V restano immutati.

Osservate come il flag di Zero non venga modificato nel caso di un risultato uguale a zero. Per eseguire operazioni aritmetiche in precisione multipla, bisogna, prima di tutto, porre a uno il flag di Zero (usando MOVE to CCR), quindi eseguire l'operazione. Se un qualunque risultato è diverso da zero, il flag di Zero sarà azzerato; altrimenti, il risultato è zero ed il flag di Zero resta a uno.

ABC (Addizione Decimale con Extend tra Valori in Memoria)

Questa istruzione somma il contenuto di una locazione di memoria ed il valore del flag di Extend (X) al contenuto di un'altra locazione di memoria. L'indirizzo dell'operando sorgente viene prelevato dal registro indirizzi sorgente, quello dell'operando destinazione dal registro indirizzi destinazione. Il risultato viene depositato nel registro indirizzi destinazione. Il contenuto di entrambi i registri indirizzi viene decrementato prima dell'operazione. L'addizione è eseguita usando l'aritmetica BCD (decimale codificata binaria). Il codice oggetto per questa forma dell'istruzione ABCD è:

La Figura 22-2 mostra l'esecuzione dell'istruzione ABCD, con il registro A1 che contiene l'indirizzo dell'operando sorgente ed il registro A4 l'indirizzo dell'operando destinazione.

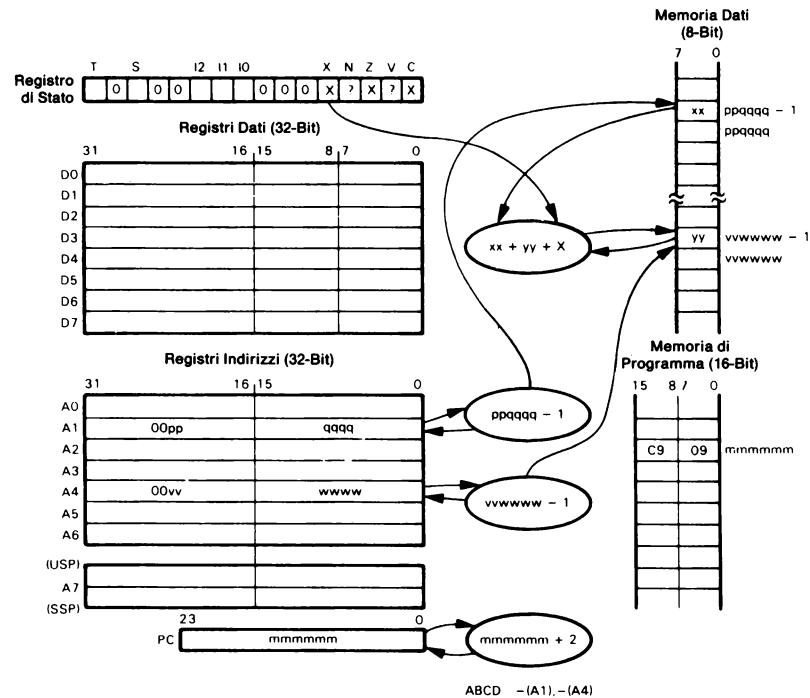



Figura 22-2. Esecuzione dell'Istruzione ABCD con Operandi in Memoria.

Come potete vedere, il contenuto di entrambi i registri indirizzi viene decrementato prima che siano prelevati gli addendi in modo da facilitare l'addizione multibyte BCD. Infatti, una stringa di cifre BCD, che rappresentano un numero decimale, viene messa in memoria in modo che le cifre meno significative occupino l'indirizzo più alto. Nel Capitolo 8 troverete una trattazione completa dell'addizione multibyte BCD.

I flag di stato sono interessati allo stesso modo della precedente istruzione ABCD.

ADD (Addizione Binaria)

Questa istruzione somma il contenuto dell'operando sorgente all'operando destinazione, salvando il risultato in quest'ultimo.

Esistono due forme generali di questa istruzione. Nella prima, è un registro dati a fornire uno degli operandi ed a contenere il risultato finale. In questo caso, sono consentiti tutti i tipi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi	X*		001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

* Non è consentito con operazioni della grandezza di un byte

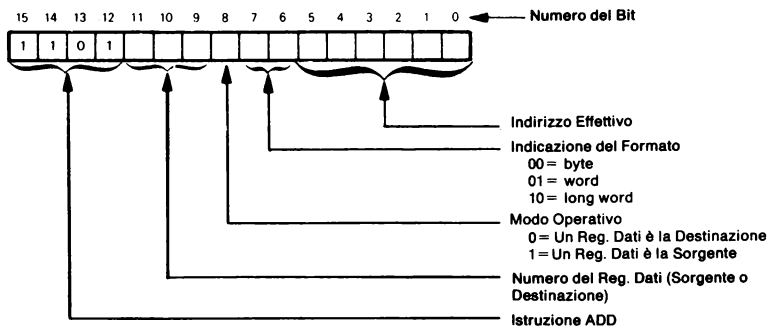
La sola limitazione è che l'indirizzamento diretto a registro indirizzi non può fornire l'operando sorgente se la sua lunghezza deve essere di un byte, in quanto i registri indirizzi non possono gestire dati di questa dimensione.

Nell'altra forma generale dell'istruzione ADD, un registro dati deve provvedere l'operando sorgente; il secondo operando, che

conterrà anche il risulato finale, può essere specificato con uno dei seguenti tipi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi		X	001	rrr
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Le limitazioni relative ai tipi di indirizzamento utilizzabili, in questo caso, sono piuttosto logiche. Infatti, molto spesso, sia i programmi che i dati immediati si trovano nella memoria di sola lettura (ROM) e, perciò, queste locazioni non potranno servire per contenere il risultato finale.

Il codice oggetto dell'istruzione ADD è:

La Figura 22-3 mostra l'esecuzione di un'istruzione ADD, con l'impiego dell'indirizzamento assoluto lungo e con D3 come registro destinazione.

I flag di Carry (C) e di Extend (X) sono posti a 1 se si verifica un riporto, in seguito all'operazione; altrimenti sono azzerati. Il flag di Zero (Z) è posto a 1 se il risultato è zero; altrimenti viene azzerato. Il flag di Negativo (N) è posto a 1 in caso di risultato negativo; altrimenti è azzerato. Il flag di Overflow (V) è posto a 1 in presenza di overflow; in caso contrario viene azzerato. Vi rimandiamo al Capitolo 8 per una discussione più ampia delle interazioni di questi flag nell'aritmetica priva di segno ed in complemento a due.

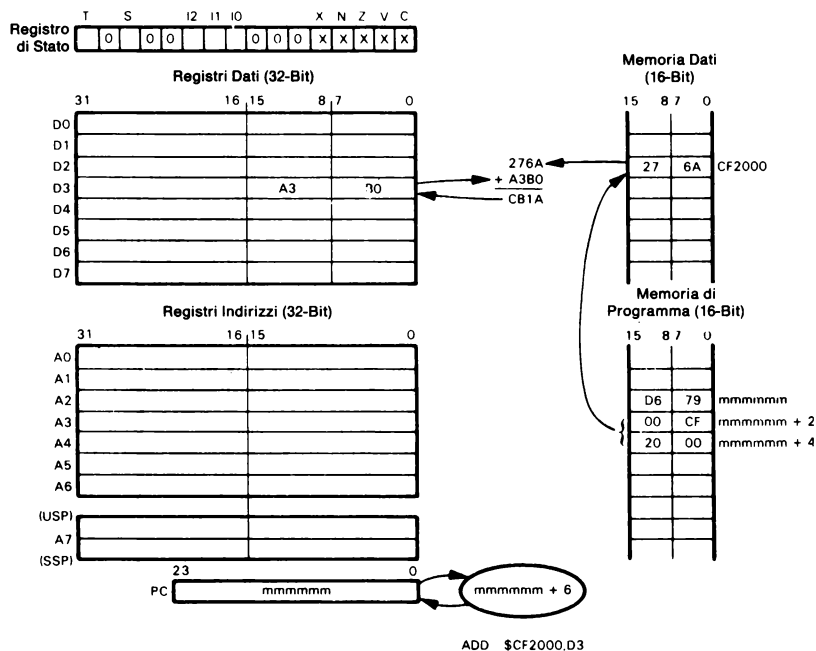
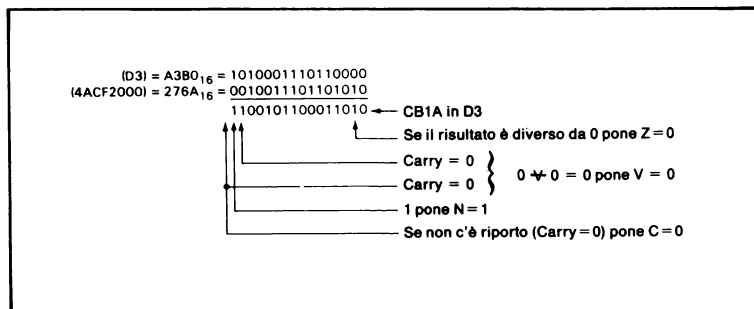



Figura22-3. Esecuzione dell'Istruzione ADD con l'impiego dell'Indirizzamento Assoluto Lungo.

Usando gli operandi della Figura 22-3, questo è ciò che accade quando viene eseguita l'istruzione ADD $CF2000,D3:

ADDA (Addizione con valore e il Risultato in un Registro Indirizzi)

Questa istruzione è un caso speciale dell'istruzione ADD e somma un operando sorgente al registro indirizzi specificato. Per l'operando sorgente sono consentiti tutti i tipi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi	X	X	001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto dell'istruzione ADDA è:

Confrontando questo codice oggetto con quello dell'istruzione ADD si vedrà che sono identici, tranne che per la dimensione del

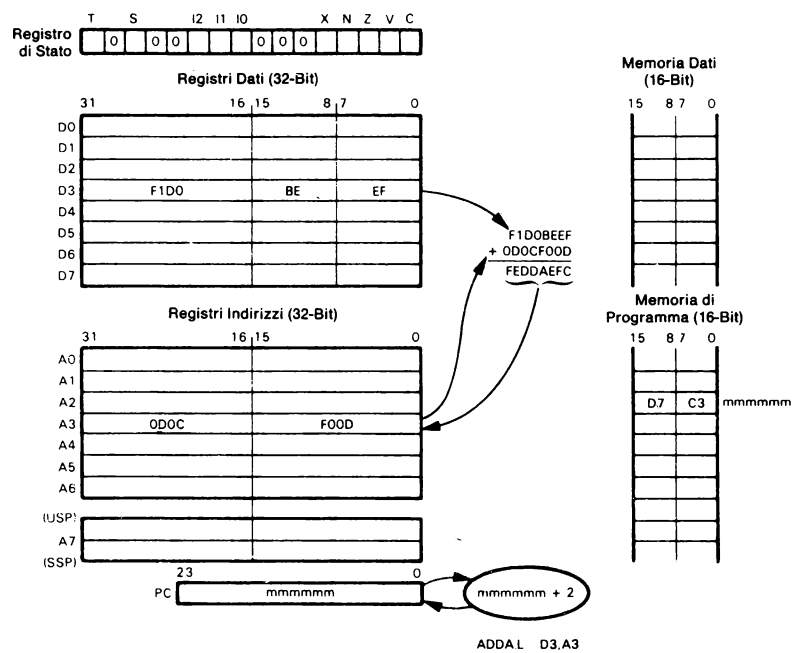



Figura 22-4. Esecuzione dell'Istruzione ADDA con l'indirizzamento Diretto a Registro.

campo: la sequenza di 2 bit che non era usata con l'istruzione ADD, è impiegata per indicare un'istruzione ADDA.

La Figura 22-4 mostra l'esecuzione dell'istruzione ADDA, con il registro D3 che fornisce l'operando sorgente a 32 bit ed il registro A3 che rappresenta il registro destinazione.

Specificando un operando sorgente a 16 bit (word), invece di una long word, l'operando sorgente viene trasformato in un operando lungo mediante l'estensione del segno e la somma è eseguita utilizzando tutti i 32 bit del registro indirizzi specificato.

Una differenza significativa fra questa istruzione e la normale istruzione ADD sta nel fatto che, in questo caso, non è interessato nessuno dei flag di stato.

ADDI (Addizione con Dato Immediato)

Questa istruzione è usata per sommare il dato immediato presente nei successivi byte di memoria all'operando destinazione specificato. Il risultato è memorizzato nell'operando destinazione. I tipi di indirizzamento utilizzabili sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi			001	rrr
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			

Il codice oggetto per questa istruzione è:

Figura 22-4. Esecuzione dell'istruzione ADDA con l'indirizzamento Diretto a Registro.

È possibile specificare se il formato dell'operazione deve essere di un byte, di una word o di una long word. Il dato immediato segue la word d'istruzione e deve corrispondere alla lunghezza specificata. Perciò, un dato immediato di una o due word seguirà il codice

operativo dell'istruzione nella memoria di programma. Se l'istruzione indica un operando di un byte, viene usato il byte di ordine basso (secondo) della word che rappresenta il dato immediato. L'assemblatore, automaticamente, seleziona il byte corretto.

La figura 22-5 mostra l'esecuzione dell'istruzione ADDI con un dato di lunghezza pari ad una word (16 bit), usando l'indirizzamento assoluto corto. Come potete vedere, la word successiva al codice operativo dell'istruzione contiene il dato immediato (in questo caso $A3B0_{10}$). L'indirizzo assoluto corto, che subisce l'estensione del segno per indicare l'operando destinazione, segue il dato immediato nella memoria di programma. Il dato immediato viene sommato al contenuto della locazione 000420_{16} ed il risultato è memorizzato nella locazione 000420_{16}.

I flag di stato X, N, Z e C sono interessati in modo analogo a quanto accade con l'istruzione ADD.

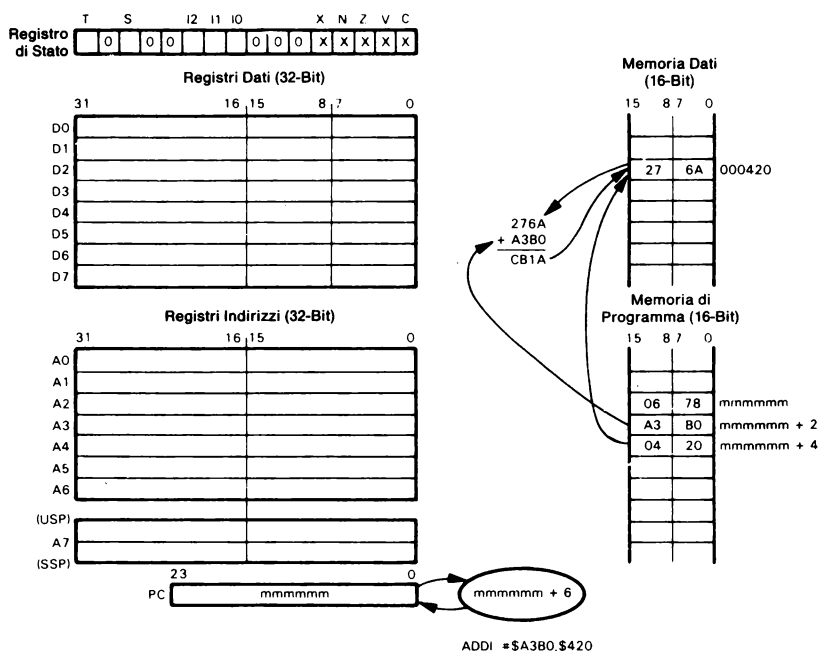



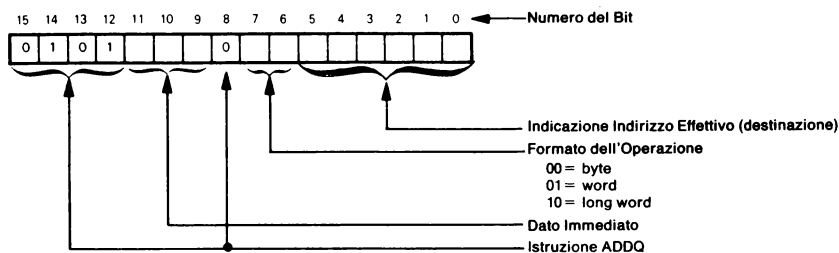
Figura 22-5. Esecuzione dell'Istruzione ADDI con l'Indirizzamento Assoluto.

ADDQ (Addizione con Dato Immediato nella Word di Codice Oggetto)

Questa istruzione somma il dato immediato, contenuto all'interno della word di codice oggetto dell'istruzione, all'operando destinazione, nella cui locazione viene salvato il risultato finale. I tipi di indirizzamento possibili sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi		X*	001	rrr
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			

* Non consentito con operazioni della grandezza di un byte

Il codice oggetto dell'istruzione ADDQ è:

I tre bit del dato immediato sono i bit 9, 10 e 11 del codice oggetto dell'istruzione. Perciò, possono essere rappresentati dati immediati compresi fra 1 e 8: quando i bit sono tutti a zero, l'assemblatore li interpreta come un dato immediato uguale a 8.

La Figura 22-6 mostra l'esecuzione dell'istruzione ADDQ, con l'indirizzamento diretto a registro dati. Quando viene eseguita l'istruzione ADDQ.B #6,D4, il valore immediato di 6 (110_2) è sommato al byte meno significativo del registro D4 ed il risultato viene messo in D4. I bit 8-31 del registro dati non sono interessati. I flag X, N, Z, V e C sono modificati al pari di quanto avviene con l'istruzione ADD.

ADDX (Addizione con Extend tra Valori nei Registri)

Questa istruzione somma il contenuto del registro dati sorgente ed il valore del flag di Extend (X) al contenuto del registro dati destinazione. Il risultato viene messo nel registro dati destinazione. Il formato dell'operazione può essere di un byte, di una word o di una long word.

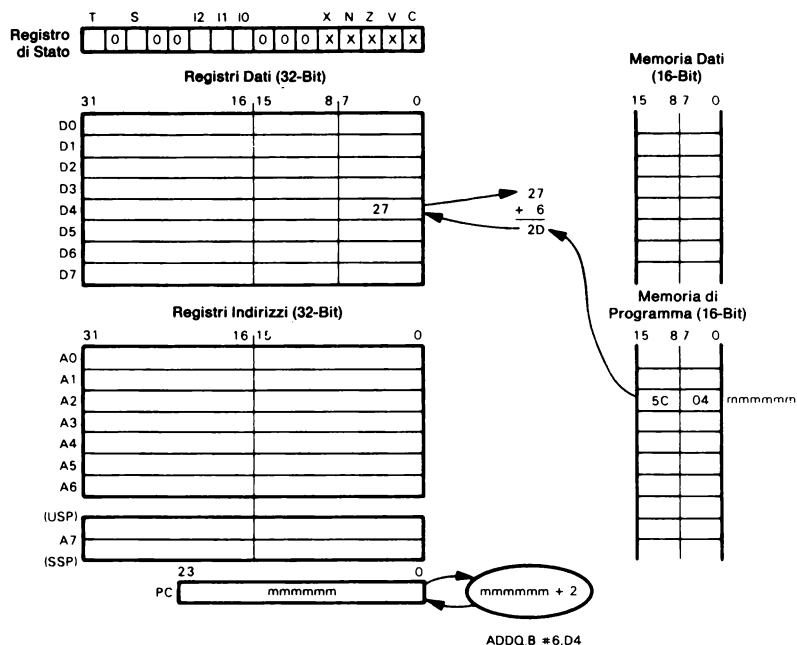
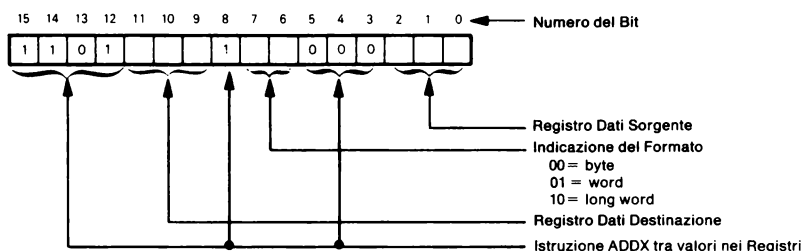
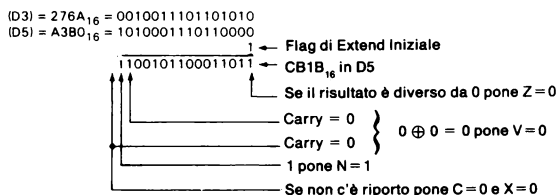



Figura 22-6. Esecuzione dell'Istruzione ADDQ con l'Indirizzamento Diretto a Registro Dati.

Il codice oggetto per questa istruzione è:

La Figura 22-7 mostra l'esecuzione dell'istruzione ADDX, con il registro D3 come registro sorgente e D5 come registro destinazione. Supponiamo che xxxx = $276A_{16}$, Extend = 1 e yyyy = $A3B0_{16}$. Dopo che il processore ha eseguito l'istruzione ADDX D3,D5, il contenuto di D5 sarà $CB1B_{16}$.

Dal momento che questa istruzione utilizza dati della lunghezza di una word, sono sommati soltanto i bit da 0 a 15 dei registri dati.

I flag di Carry (C) e di Extend (X) sono posti a 1 se nell'operazione viene generato un riporto e sono azzerati in caso contrario. Il flag di Zero viene azzerato se il risultato è diverso da zero; altrimenti resta immutato. Il flag di Negativo (N) è posto a 1 se il risultato è negativo, altrimenti è azzerato. Il flag di Overflow (V) è posto a 1 in caso di overflow; altrimenti viene azzerato.

Osservate come il flag di Zero non venga modificato nel caso di un risultato uguale a zero. Nell'aritmetica in precisione multipla bisogna, prima di tutto, porre a uno il flag di Zero (usando MOVE to CCR), quindi eseguire l'operazione. Se un qualunque risultato è diverso da zero, il flag di Zero sarà azzerato; altrimenti, il risultato è zero ed il flag di Zero resta a uno.

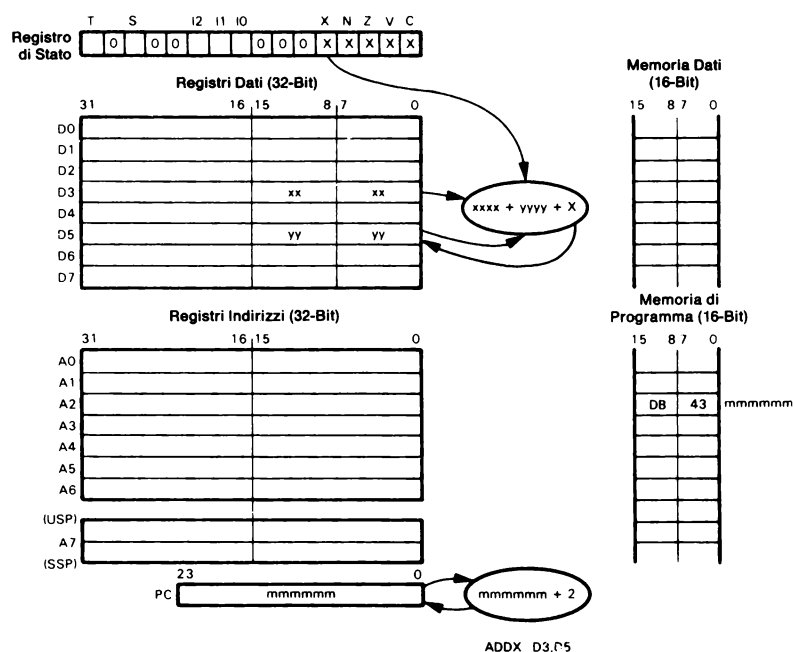
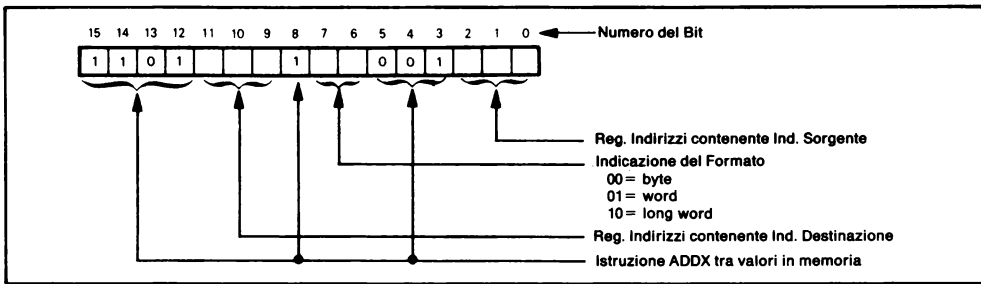



Figura 22-7. Esecuzione dell'Istruzione ADDX da Registro a Registro.

ADDX (Addizione con Extend tra Valori in Memoria)

Questa istruzione somma il contenuto di una locazione di memoria ed il valore del flag di Extend (X) al contenuto di un'altra locazione di memoria. L'indirizzo di memoria dell'operando sorgente è contenuto in un registro indirizzi e l'indirizzo relativo all'operando destinazione in un altro registro indirizzi. Prima dell'operazione, il contenuto di entrambi i registri indirizzi viene decrementato.

Il codice oggetto per questa forma dell'istruzione ADDX è:

Il dato può avere la lunghezza di un byte, di una word o di una long word.

La figura 22-8 mostra l'esecuzione dell'istruzione ADDX, con il registro A1 che contiene l'indirizzo dell'operando sorgente in memoria ed il registro A4 quello dell'operando destinazione.

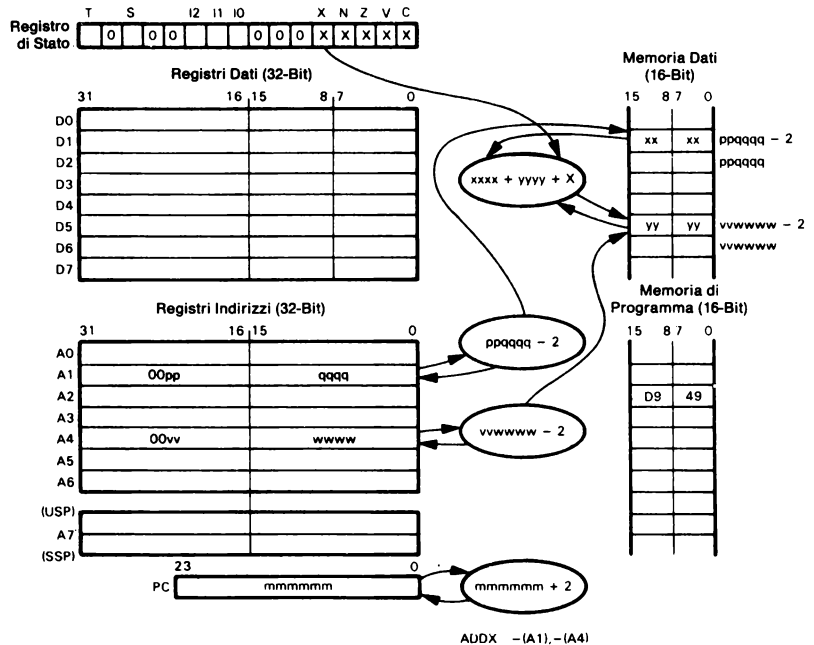



Figura 22-8. Esecuzione dell'Istruzione ADDX tra valori in Memoria.

Come potete notare, il contenuto di entrambi i registri indirizzi viene decrementato di 2, prima che gli operandi siano utilizzati per la somma. Se fosse stata specificata una lunghezza del dato pari ad un byte, allora i registri indirizzi sarebbero stati decrementati di 1; se, invece, il dato avesse avuto le dimensioni di una long word, il decremento sarebbe stato di 4. Questo predecremento facilita l'aritmetica binaria in precisione multipla, in quanto i registri indirizzi sono modificati per accedere automaticamente al successivo byte, word o long word da utilizzare. Vi rimandiamo al Capitolo 8 per

una trattazione più completa dell'aritmetica in precisione multipla.
I flag di stato sono interessati in modo analogo a quanto avviene con l'istruzione ADDX tra valori contenuti nei registri.

AND (AND Logico)

Questa istruzione esegue un AND logico, bit per bit, del contenuto dell'operando sorgente con quello dell'operando destinazione, salvando il risultato nella locazione di destinazione.

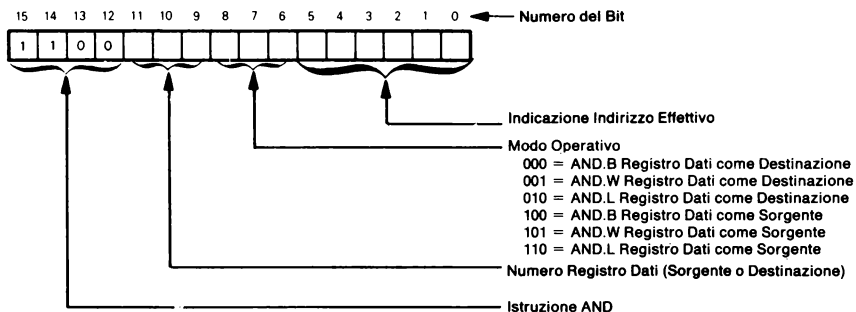
Ne esistono due forme generali. Nella prima è un registro dati a fornire l'operando destinazione e sono consentiti tutti i tipi di indirizzamento per l'operando sorgente, tranne quello diretto a registro indirizzi:

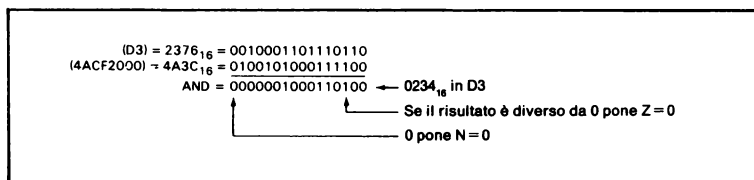
	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Nell'altra forma dell'istruzione AND è un registro dati a fornire l'operando sorgente; l'operando destinazione può essere indicato con uno dei seguenti modi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi		X	001	rrr
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Figura 22-9. Esecuzione dell'Istruzione AND con Indirizzamento Assoluto Lungo.

Il codice oggetto dell'istruzione AND è:

La Figura 22-9 mostra l'esecuzione dell'istruzione AND, utilizzando l'indirizzamento assoluto lungo e con D3 come registro destinazione. Con gli operandi della Figura 22-9. questo è ciò che accade quando viene eseguita l'istruzione AND $4ACF2000,D3:

```
        (D3) = 2376₁₆ = 0010001101110110
(4ACF2000) = 4A3C₁₆ = 0100101000111100
                AND = 0000001000110100  ←— 0234₁₆ in D3
                      ↑             ↑
                      |             └—— Se il risultato è diverso da 0 pone Z=0
                      └———————————————— 0 pone N=0
```

Il flag N sarà posto a 1 se il bit più significativo del risultato è uguale a 1, altrmenti sarà azzerato. Il flag di stato Z diventa 1 se il risultato è zero e 0 in caso contrario. I flag di Stato C e V sono sempre azzerati in seguito ad un AND. Il flag di stato X resta immutato.

ANDI (AND su un Dato Immediato)

Questa istruzione è usata per eseguire l'AND di un dato immediato, presente nelle locazioni di memoria successive all'operando destinazione, nella cui locazione viene, poi, salvato il risultato. I tipi di indirizzamento possibili sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			
Registro di Stato		X	111	100

Si noti come l'operando destinazione possa essere rappresentato dai codici di condizione o dall'intero registro di stato. **Se come destinazione abbiamo l'intero registro di stato, questa diventerà un'istruzione privilegiata e potrà essere eseguita solo quando il processore si trova nel modo Supervisore.**

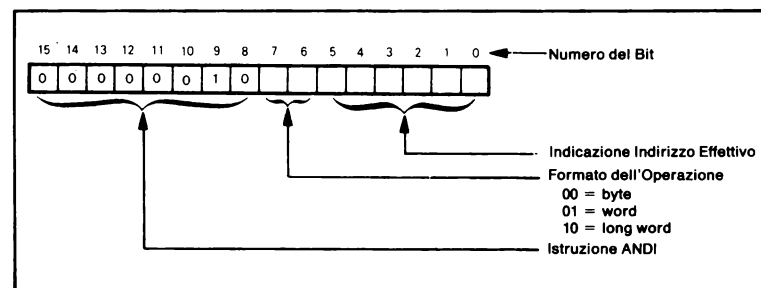
Il codice operativo per l'istruzione ANDI è:

La lunghezza dell'operazione può essere di un byte, di una word o di una long word. Il dato immediato segue la word d'istruzione in memoria e deve corrispondere alla lunghezza specificata. Perciò, una o due word di dato immediato devono seguire il codice operativo nella memoria di programma. Se l'istruzione specifica un operan-

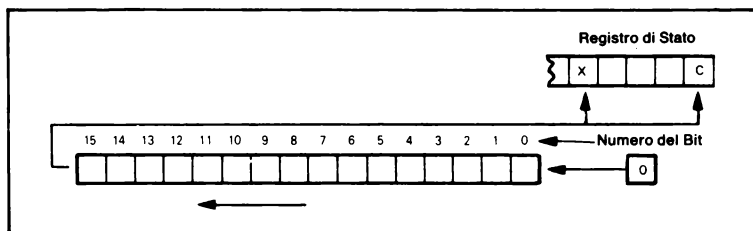
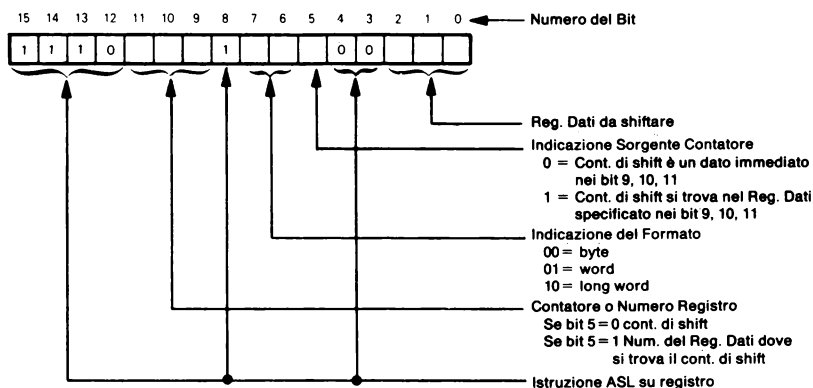
Figura 22-10. Esecuzione dell'Istruzione ANDI con l'Indirizzamento Assoluto.

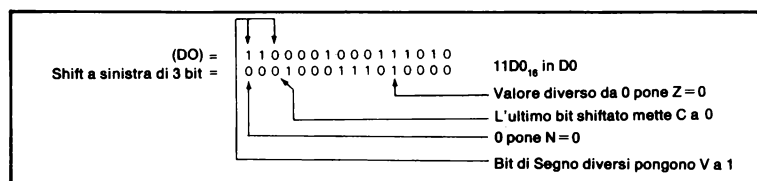
do della lunghezza di un byte, viene usato il byte di ordine basso (secondo) della word di dato immediato. È l'assemblatore che provvede, automaticamente, a prendere il byte giusto. Se l'istruzione si riferisce al registro di stato ed il formato dell'operazione è di un byte, l'operando destinazione è il byte di ordine basso del registro di stato e sono interessati solo i codici di condizione. Se la lunghezza dell'operazione è pari ad una word, l'operando destinazione è l'intero registro di stato e si tratterà di un'istruzione privilegiata.

La Figura 22-10 mostra l'esecuzione dell'istruzione ANDI, con un formato del dato pari ad una word (16 bit) e facendo uso dell'indirizzamento assoluto corto. Come potete vedere, la word successiva al codice operativo dell'istruzione contiene il dato immediato (2376_{16} in questo esempio). L'indirizzo assoluto corto, che subisce l'estensione del segno per indicare l'operando destinazione, segue il dato immediato nella memoria di programma. Viene eseguito un AND, bit per bit, del dato immediato con il contenuto della locazione 000420_{16} ed il risultato (0234_{16}) è posto nella locazione 000420_{16}.

I flag di stato N e Z sono modificati dall'istruzione ANDI, a seconda del risultato ottenuto. I flag di stato V e C sono sempre azzerati. Il flag di stato X resta immutato. Naturalmente, se la destinazione dell'istruzione sono i codici di condizione o l'intero registro di stato, allora i codici di condizione sono modificati in accordo con il risultato dell'operazione stessa.

ASL (Shift Aritmetico a Sinistra in un Registro Dati)

Questa istruzione esegue lo shift aritmetico a sinistra del contenuto del registro dati specificato. In seguito allo shift, i flag di Carry e di Extend ricevono l'ultimo bit uscito dal registro dati, mentre uno zero viene inserito nel bit di ordine basso, come appare dallo schema seguente:

Il numero di shift può essere specificato dal contenuto di un altro registro dati o da un dato immediato. Nel caso che sia impiegato un registro dati, sono i sei bit meno significativi di questo registro ad indicare il numero di shift (sono possibili valori compresi fra 0 e 63). Usando un dato immediato, si possono specificare valori compresi fra 1 e 8 all'interno del codice operativo dell'istruzione. Un valore zero equivale ad uno shift di 8. Il codice oggetto dell'istruzione ASL a registro è:

Il bit 5 del codice operativo dell'istruzione stabilisce se il numero di shift verrà indicato mediante un dato immediato, contenuto nei bit 9, 10 e 11,oppure se esso si trova in un altro registro dati, il cui numero è indicato sempre dai bit 9, 10 e 11.

La Figura 22-11 mostra l'esecuzione di un'istruzione ASL, con il registro D0 come operando destinazione e con il numero di shift contenuto nel registro D3. Se il registro D0 inizialmente contiene C23A16 e D3 contiene 03, verrà eseguito uno shift di 3 bit a sinistra, come appare nello schema seguente:

I bit di ordine alti, usciti dal registro in seguito ai vari shift, vanno sia nel flag di Carry che in quello di Extend. Se il numero di shift è zero, il flag di Carry è azzerato, mentre il Flag di Extend resta invariato. Degli zeri finiscono nei bit di ordine basso del registro dati.

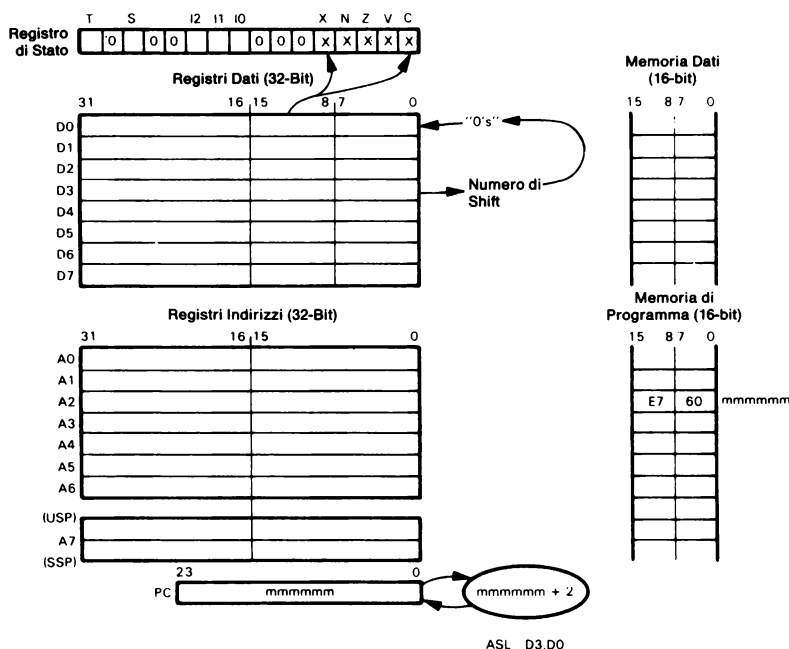



Figura 22-11. Esecuzione dell'Istruzione ASL con Operando e Numero di Shift nei Registri Dati.

Il flag di Overflow (V) indica il verificarsi di un eventuale cambiamento di segno durante lo shift. I flag N e Z sono modificati in base al valore del risultato.

La Figura 22-11 mostrava un'operazione di ASL con un operando della grandezza di una word nel registro dati. Soltanto i bit da 0 a 15 del registro dati erano interessati; i bit da 16 a 31 restavano immutati. L'istruzione ASL può anche agire su operandi di un byte o di una long word, contenuti sempre in un registro dati. Per quanto riguarda l'istruzione descritta nella Figura 22-11, si potevano ottenere gli stessi risultati con l'istruzione seguente:

```
ASL #3,D0
```

Questa istruzione utilizza un valore immediato di 3 per indicare il numero di shift, invece del contenuto di un altro registro dati (D3) come accadeva nell'esempio precedente. Il vantaggio di usare un registro dati per indicare il numero di shift sta nel fatto di poterne modificare il contenuto durante l'esecuzione del programma. Un valore immediato è contenuto, invece, nella memoria di sola lettura e, quindi, non è modificabile.

ASL (Shift Aritmetico a Sinistra in Memoria)

Questa istruzione esegue le stesse operazioni dell'istruzione precedente, ma utilizza un operando contenuto in memoria, invece di in un registro dati. Questa versione dell'istruzione ASL presenta due restrizioni: il formato dell'operando è limitato ad una word e lo shift può essere solo di un bit.

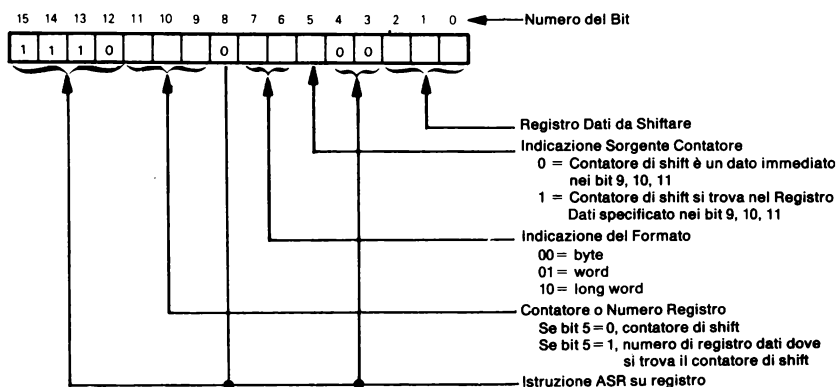
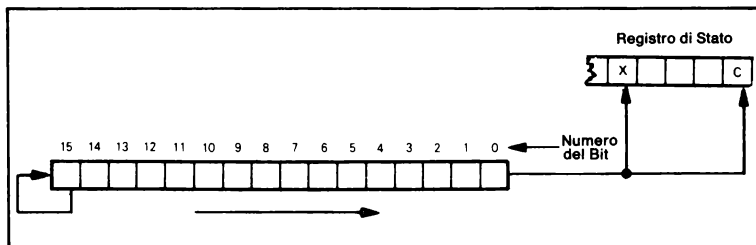
Gli indirizzamenti possibili con l'istruzione ASL sulla memoria sono:

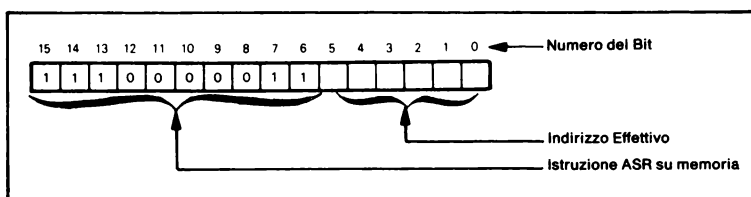
	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postinci emento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

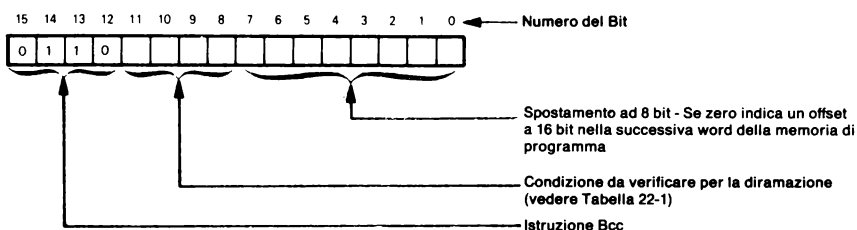
Il codice oggetto di questa istruzione è:

L'esecuzione di questa istruzione ASL è analoga a quanto mostrato per la versione con registro dati. L'operando in memoria subisce uno shift a sinistra di un bit ed il bit di ordine alto (bit 15) va ad occupare i flag di Carry e di Extend del registro di stato. Nel bit di ordine basso della word in memoria viene inserito uno zero. Il bit di Overflow (V) indicherà un eventuale cambiamento di segno avvenuto in seguito allo shift.

Questa istruzione esegue lo shift aritmetico a destra dei bit dell'operando. Analogamente a quanto accade con le istruzioni ASL, descritte nelle pagine precedenti, anche in questo caso esistono due versioni. La prima permette di eseguire lo shift di un operando, contenuto in un registro dati, del numero di posizioni specificate dal contenuto di un altro registro dati oppure da un valore immediato indicato nella stessa word d'istruzione. La seconda versione dell'istruzione ASR consente lo shift di un bit di una word contenuta in memoria. I tipi di indirizzamento possibili sono gli stessi mostrati per le istruzioni ASL sulla memoria e su un registro.

Il codice oggetto per l'istruzione ASR su un registro è:

Quando viene eseguita l'istruzione ASR, l'operando subisce uno shift a destra del numero di posizioni indicate, mentre i bit di ordine basso, usciti in seguito allo shift, vanno nei flag di Carry e di Extend del registro di stato. Se il numero di shift è zero, il flag di Carry è azzerato, mentre il flag di Extend resta invariato. Il bit di ordine alto (segno) si sposta a destra di una posizione e, in questo caso, viene anche duplicato:

Come nel caso dell'istruzione ASL, la differenza fra le versioni su un registro sulla memoria è dovuta al fatto che quest'ultima deve avere un operando della grandezza di una word e lo shift può essere

solo di una posizione. Il codice oggetto dell'istruzione ASR sulla memoria è:

Bcc (Diramazione Condizionata)

Questa istruzione trasferisce il controllo del programma ad un indirizzo specificato relativamente al valore del contatore di programma, solo se viene soddisfatta la condizione specificata da cc. L'unico indirizzamento possibile con questa istruzione è quello relativo al contatore di programma. Lo spostamento rispetto alla locazione dell'istruzione può essere compreso fra -126 e + 129 o fra -32766 e +32769. Il codice oggetto è:

I bit da 8 a 11 della word d'istruzione indicano la condizione che deve essere controllata per stabilire se effettuare o meno la diramazione. **La Tabella 22-1 elenca le condizioni possibili con questa istruzione e indica quali flag di stato sono utilizzati per stabilire se il test è riuscito o meno.**

Se il risultato ottenuto è uno, allora la condizione specificata è soddisfatta. Viene calcolato un nuovo valore per il contatore di programma, sommando il valore di spostamento ad 8 bit (con estensione del segno), contenuto nella word d'istruzione, al contatore di programma, dopo che questo è stato incrementato di due. Se gli otto bit meno significativi della word dell'istruzione B_{cc} sono zero, il nuovo valore del contatore di programma viene calcolato utilizzando un valore di spostamento di 16 bit, contenuto nella word successiva a quella d'istruzione. L'esecuzione continua a partire dalla locazione indicata dal nuovo valore del contatore di programma.

Se il risultato del test è zero, significa che la condizione specificata non è stata soddisfatta e l'esecuzione continuerà con l'istruzione successiva.

La Figura 22-12 mostra l'esecuzione dell'istruzione BVC. Se il flag di Overflow (V) del registro di stato vale 1, l'esecuzione del program-

ma prosegue con l'istruzione immediatamente successiva. Mentre, se il flag di Overflow (V) è azzerato, il valore di spostamento ad 8 bit (in complemento a due), presente nel codice d'istruzione, viene sommato al contenuto del contatore di programma, dopo che questo è stato incrementato di due. Con un valore di spostamento pari a 40_{16}, come nel caso della Figura 22-12, la diramazione, se ha luogo, trasferisce il controllo alla word di memoria mmmmmm + 42_{16}.

Tabella 22-1. Test di Condizione con l'Istruzione Bcc.

Mnemonico (cc)	Condizione	Campo Condizione	Test
HI	Alto	0010	$\bar{C} \wedge \bar{Z}$
LS	Basso o Uguale	0011	$C \vee Z$
CC	Carry=0	0100	$\bar{C}$
CS	Carry=1	0101	C
NE	Diverso	0110	$\bar{Z}$
EQ	Uguale	0111	Z
VC	Non Overflow	1000	$\bar{V}$
VS	Overflow	1001	V
PL	Più	1010	$\bar{N}$
MI	Meno	1011	N
GE	Maggiore o Uguale	1100	$(N \wedge V) \vee (\bar{N} \wedge \bar{V})$
LT	Minore	1101	$(N \wedge \bar{V}) \vee (\bar{N} \wedge V)$
GT	Maggiore	1110	$(N \wedge V \wedge \bar{Z}) \vee (\bar{N} \wedge \bar{V} \wedge \bar{Z})$
LE	Minore o Uguale	1111	$Z \vee (N \wedge \bar{V}) \vee (\bar{N} \wedge V)$

Figura 22-12. Esecuzione dell'Istruzione BVC.

Osservate come il valore di spostamento, presente nel codice oggetto, venga sommato al contatore di programma, solo dopo che

questo è stato incrementato di due. Nel linguaggio assembly, il simbolo dell'asterisco si riferisce alla locazione dell'istruzione in cui esso compare ed anche **il valore di spostamento è relativo alla locazione dell'istruzione.** L'assemblatore lo riduce, automaticamente, di due quando genera il codice oggetto. **Naturalmente, l'impiego di valori di spostamento assoluti è una cattiva abitudine; è consigliabile utilizzare sempre delle label.**

Infatti, solitamente non viene usata la forma dell'istruzione mostrata nella Figura 22-12 (BVC * + $42). Si preferisce far ricorso ad una label, che indichi il punto dove deve proseguire l'esecuzione del programma, qualora venga soddisfatta la condizione specificata. Provvederà l'assemblatore a stabilire lo spostamento necessario per raggiungere la locazione corrispondente ed a fornire l'opportuno valore a 8 o 16 bit, che sarà contenuto nel codice oggetto dell'istruzione.

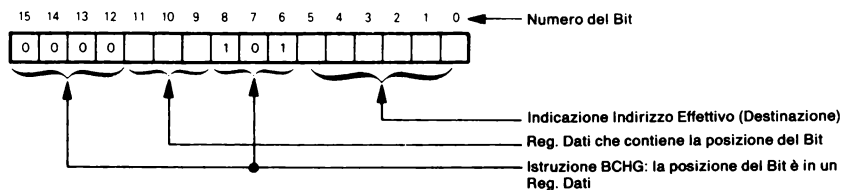
L'istruzione B_{cc} non modifica nessuno dei flag di stato. Il precedente valore del contatore di programma va perso.

BCHG (Test e Complemento di un Bit)

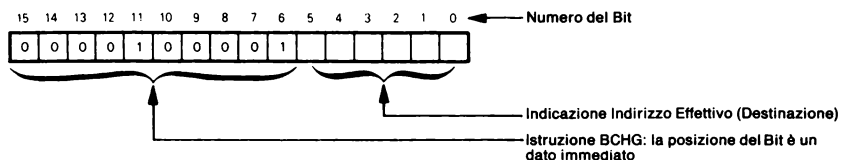
Questa istruzione controlla un determinato bit di una locazione di memoria o di un registro dati, il cui valore viene riflesso nel flag di Zero (Z). Dopo questa operazione di controllo, lo stato del bit, nel registro dati o nella locazione di memoria, è complementato.

Di questa istruzione esistono due forme. Nella prima, il numero del bit da controllare è contenuto in un registro dati. Nell'altra, il numero del bit viene specificato mediante un dato immediato nella word successiva a quella dell'istruzione. L'operando destinazione, che contiene il bit da controllare, può essere specificato con uno dei tipi di indirizzamento seguenti:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X		rrr
Indiretto a Registro con Postincremento		X		rrr
Indiretto a Registro con Predecremento		X		rrr
Indiretto a Registro con Spostamento		X		rrr
Indiretto a Registro con Indice		X		rrr
Assoluto Corto		X		000
Assoluto Lungo		X		001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			

Quando viene impiegato un registro dati per indicare il numero del bit, il codice oggetto dell'istruzione BCHG è:

I bit 9, 10 e 11 della word d'istruzione specificano il registro dati contenente il numero del bit da controllare. Se quest'ultimo si trova in un altro registro dati, allora è possibile controllare uno qualunque dei 32 bit. In tal caso, il numero del bit è indicato mediante i sei bit meno significativi (modulo 32) del primo registro dati. Se il bit da controllare si trova in memoria, allora il formato dell'operazione BCHG non può essere superiore ad un byte. In questo caso, il numero del bit da controllare all'interno del byte di memoria è indicato dai tre bit meno significativi (modulo 8) del registro dati.

La Figura 22-13 mostra l'esecuzione di un'istruzione BCHG, con indirizzamento diretto a registro dati. Il registro dati D4 contiene la posizione del bit da controllare (bit numero 3) nel registro dati D1. Dopo l'esecuzione di questa istruzione, il registro dati D1 conterrà tutti zeri, in quanto il bit 3, una volta controllato, viene complementato.

Nella seconda forma di questa istruzione, il numero del bit è indicato da un valore immediato. Il relativo codice oggetto è il seguente:

In questo caso, la posizione del bit è specificata dal dato immediato contenuto nella word che, nella memoria di programma, segue quella dell'istruzione. L'operando destinazione, contenente il bit da controllare, può essere un registro dati a 32 bit oppure un byte ad 8 bit in memoria, proprio come per la prima forma dell'istruzione. Perciò, gli stessi risultati dell'istruzione mostrata in Figura 22-13 si potevano ottenere anche nel modo seguente:

```
BCHG #3,D1
```

Per indicare il bit da controllare, questa istruzione si serve di un valore immediato 3, invece del contenuto di un altro registro dati (D4), come nel caso precedente. Il vantaggio derivante dall'impiego di un registro dati consiste nella possibilità di modificarne il contenuto durante l'esecuzione del programma, mentre non è possibile

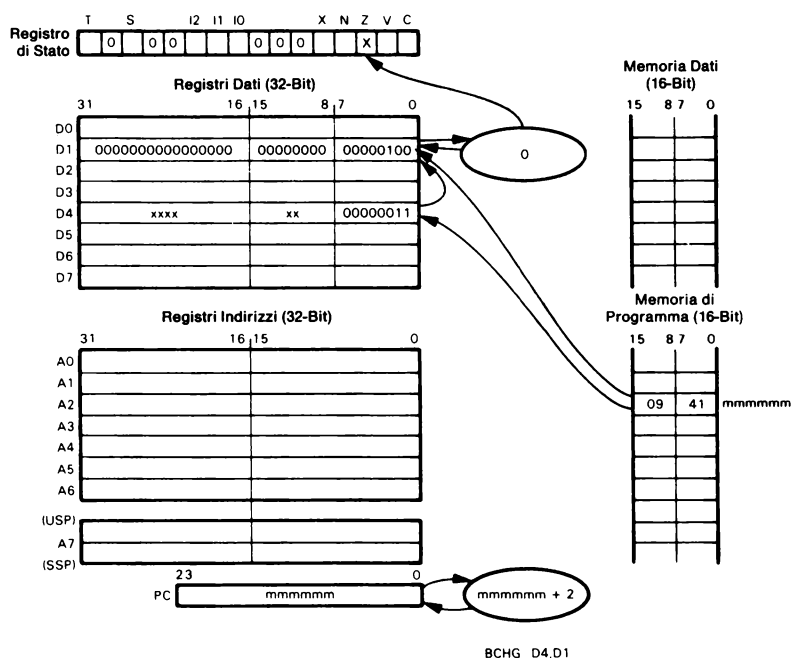



Figura 22-13. Esecuzione dell'Istruzione BCHG con Indirizzamento Diretto a Registro Dati.

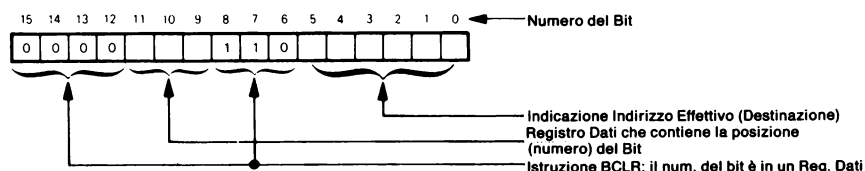
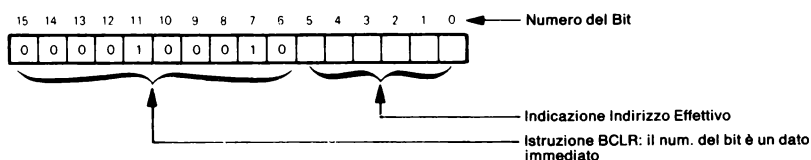
cambiare un dato immediato poichè si trova nella memoria di programma (ROM). L'unico flag di stato interessato dall'istruzione BCHG è il flag di Zero (Z), che è posto a uno se il bit controllato era 0, mentre, in caso contrario, viene azzerato.

BCLR (Test e Azzeramento di un Bit)

Questa istruzione controlla un bit di un registro dati o di una locazione di memoria e modifica il flag di Zero (Z), sulla base del suo valore. Dopo il test, il bit specificato viene azzerato. È un'istruzione simile a quella BCHG, tranne per il fatto che il bit, una volta controllato, invece di essere complementato viene sempre messo a zero.

I tipi di indirizzamento utilizzabili per indicare l'operando destinazione sono gli stessi dell'istruzione BCHG:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			

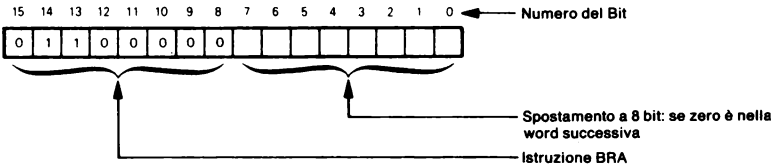
Come nel caso dell'istruzione BCHG, ci sono due forme per l'istruzione BCLR. Nella prima, la posizione del bit da controllare è contenuta in un registro dati. Il relativo codice oggetto è il seguente:

Nella seconda forma, per indicare la posizione del bit da controllare viene usato un dato immediato, contenuto nella word successiva a quella dell'istruzione:

Entrambe le versioni sono analoghe a quelle dell'istruzione BCHG, alla quale vi rimandiamo.

L'istruzione BCLR influisce solo sul flag Z, che diventa uno se il bit controllato è 0; altrimenti viene azzerato.

BRA (Diramazione Incondizionata)

Questa istruzione provoca sempre una diramazione all'indirizzo specificato, mettendo quest'ultimo nel contatore di programma. L'indirizzo è ottenuto mediante la somma del contatore di programma, dopo che è stato incrementato di due, e di un valore di sposta-

mento. Questo è ottenuto estendendo il segno di un numero in complemento a due contenuto nel byte meno significativo della word d'istruzione oppure, se questo byte contiene tutti zeri, nella word che segue il codice oggetto dell'istruzione.

Il codice oggetto per l'istruzione BRA è:

La Figura 22-14 mostra l'esecuzione dell'istruzione BRA.

Osservate come l'offset, presente nel codice oggetto, venga sommato al valore del contatore di programma dopo che questo è stato incrementato di due. Nel linguaggio assembly, il simbolo dell'asterisco si riferisce alla locazione dell'istruzione in cui esso compare ed anche **il valore di spostamento è relativo alla locazione dell'istruzione.** L'assemblatore lo riduce, automaticamente, di due quando genera il codice oggetto. **Naturalmente, l'impiego di valori di spostamento assoluti è una cattiva abitudine; è consigliabile utilizzare sempre delle label.** Considerate la seguente sezione di un programma:

```
      BRA   NEXT
      LSL   #1,D0
      -
      -
      -
NEXT  NOT   D0
```

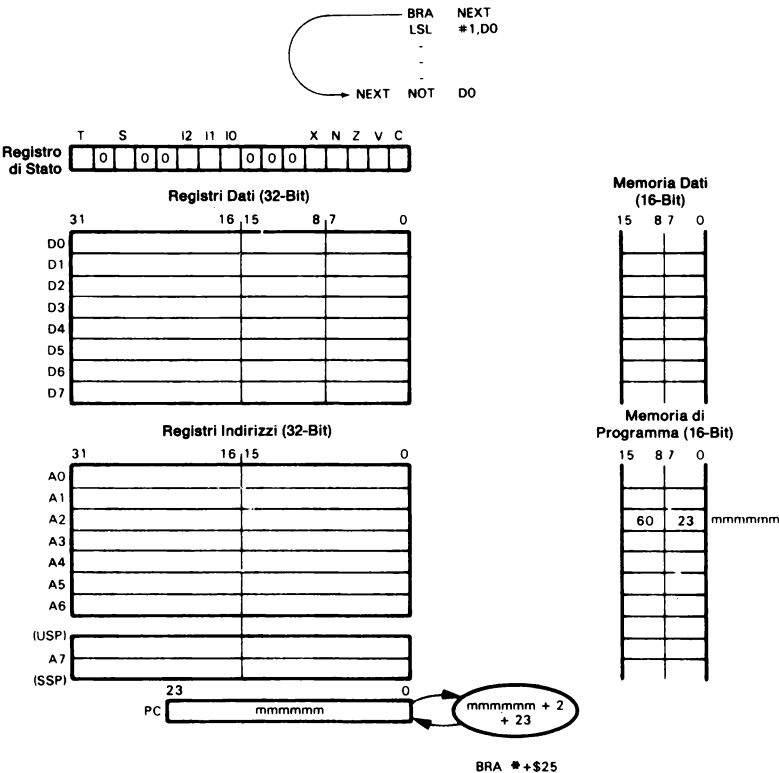



Figura 22-14. Esecuzione dell'Istruzione BRA.

Dopo l'istruzione BRA, il processore esegue sempre l'istruzione NOT. Non eseguirà mai l'istruzione LSL, a meno che un'istruzione di diramazione o di salto, in un'altra parte del programma, trasferisca ad essa il controllo.

L'effetto complessivo dell'istruzione BRA è:

PC = PC + 2 + spostamento nel codice oggetto

Il 2 è dovuto ai due byte occupati dalla stessa istruzione BRA. Il contatore di programma è incrementato appena viene prelevata l'istruzione.

L'istruzione BRA non modifica nessuno dei flag di stato. Il precedente valore del contatore di programma va perso.

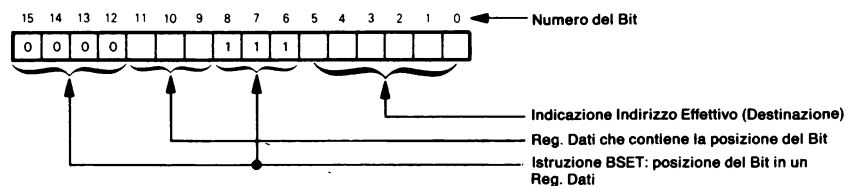
BSET (Test e Assegnazione del Valore 1 ad un Bit)

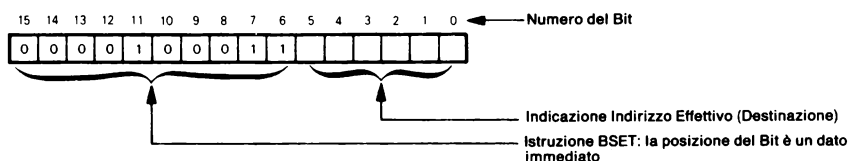
Questa istruzione controlla un determinato bit di una locazione di memoria o di un registro dati, il cui valore viene riflesso nel flag di Zero (Z). Dopo questa operazione di controllo, questo bit diventa 1.

È un'istruzione identica a quella BCHG, descritta in precedenza, tranne per il fatto che il bit controllato, in questo caso, viene sempre messo a 1 invece di essere complementato.

Di questa istruzione esistono due forme. Nella prima, il numero del bit da controllare è contenuto in un registro dati. Nell'altra, il numero del bit viene specificato mediante un dato immediato, contenuto nella word successiva a quella dell'istruzione. L'operando destinazione, che contiene il bit da controllare, può essere specificato con uno dei seguenti tipi di indirizzamento:

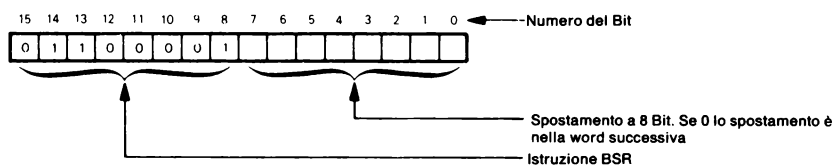
	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Num. Bit	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			

Quando viene usato un registro dati per specificare la posizione di un bit, il codice oggetto per l'istruzione BSET è:

L'esecuzione è sostanzialmente analoga a quella che abbiamo descritto per l'istruzione BCHG, tranne il fatto che il bit controllato dell'operando è sempre posto a uno. Vi rimandiamo alla descrizione di quella istruzione ed alla Figura 22-13 per ulteriori chiarimenti in merito alle modalità di esecuzione.

Nella seconda forma dell'istruzione BSET, la posizione del bit da controllare è specificata mediante un dato immediato. Il codice oggetto di questa versione è il seguente

Ancora una volta, l'esecuzione è sostanzialmente la stessa mostrata per l'istruzione BCHG, tranne per il fatto che il bit controllato diventa sempre uno. Vi rimandiamo ancora una volta a quella istruzione per maggiori dettagli. Il solo flag di stato modificato, in seguito all'istruzione BSET, è il flag di Zero (Z), che diventa uno se il bit controllato è 0; altrimenti viene azzerato.

BSR (Diramazione ad una Subroutine)

Questa istruzione mette allo stack di sistema l'indirizzo dell'istruzione immediatamente successiva. Il valore di spostamento, fornito insieme con l'istruzione BSR, viene quindi sommato al valore del contatore di programma, dopo che questo è stato incrementato di due. L'esecuzione continua a partire da questa nuova locazione.

Il codice oggetto dell'istruzione BSR è:

L'offset è un intero in complemento a due che indica lo spostamento relativo in byte, rispetto al contenuto del contatore di programma. Può trattarsi di un valore ad 8 o 16 bit. Se quella parte della word d'istruzione, relativa allo spostamento ad 8 bit, è zero, significa che un valore di spostamento a 16 bit si trova nella word successiva a quella dell'istruzione BSR. Il valore del contatore di programma, che viene sommato all'offset, è quello corrispondente alla locazione dell'istruzione più due.

La Figura 22-15 mostra l'esecuzione dell'istruzione BSR, con un valore di spostamento a 16 bit (110_{16}). Nella Figura 22-15 si presume che il programma sia eseguito in modo Utente e, perciò, il puntatore sullo stack utilizzato sarà quello relativo allo stack Utente (USP), nel registro A7. Dopo che il contatore di programma è stato incrementato per accedere all'istruzione ed all'offset a 16 bit, il nuovo valore in esso contenuto (mmmmmm + 4) viene messo allo stack di sistema. Si noti che il contenuto del puntatore allo stack (wwvvvv) è decrementato di due, una prima volta per contenere la metà di ordine alto del contatore di programma e, quindi, ancora di due per la metà di ordine basso. Perciò, il valore contenuto nel puntatore allo stack Utente, una volta eseguita l'istruzione, sarà wwvvvv-4.

Dopo che il contatore di programma è stato incrementato e messo allo stack, in esso verrà posto l'indirizzo della subroutine da eseguire.

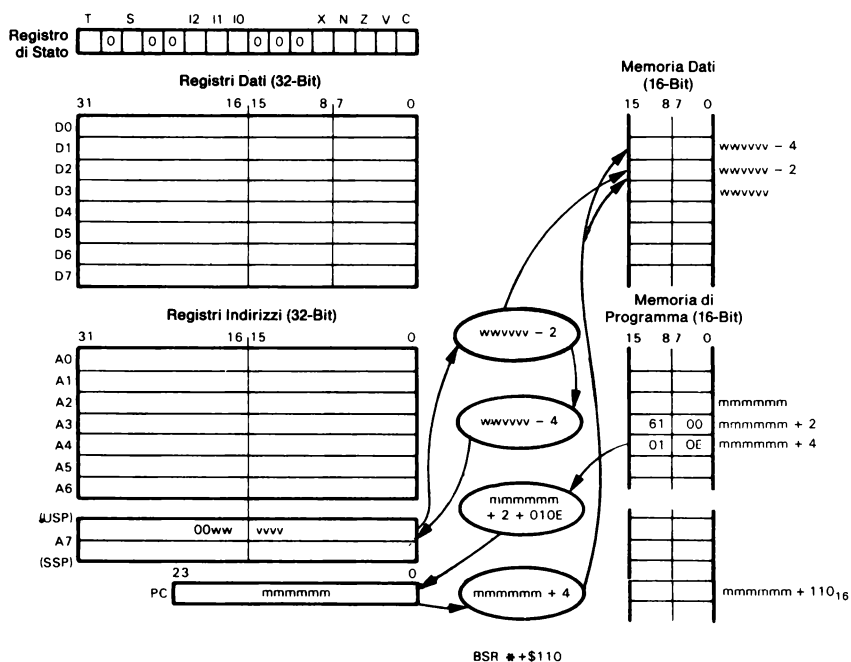



Figura 22-15. Esecuzione dell'Istruzione BSR.

BSR è analoga a BRA, tranne per il fatto che BSR salva il vecchio valore del contatore di programma sullo stack, garantendo così un legame con la subroutine. Un'istruzione RTS, posta alla fine di questa, restituisce il controllo all'istruzione immediatamente successiva a BSR, ammesso che la subroutine non abbia modificato l'indirizzo di ritorno o il contenuto del puntatore allo stack. BSR consente un salto relativo e incondizionato ad una subroutine, a differenza di quello assoluto dell'istruzione JSR. Tuttavia, si noti come JSR * + 300 abbia esattamente lo stesso effetto di BSR * + 300.

Il valore di spostamento, contenuto nel codice oggetto, viene sommato al valore del contatore di programma, una volta che

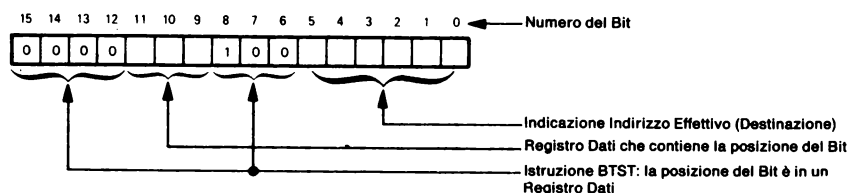
questo è stato incrementato di due. Nel linguaggio assembly, il simbolo dell'asterisco si riferisce alla locazione dell'istruzione in cui esso compare ed anche **il valore di spostamento è relativo alla locazione dell'istruzione.** L'assemblatore lo riduce automaticamente di due, quando genera il codice oggetto. **Naturalmente, l'impiego di valori di spostamento assoluti è una cattiva abitudine; è consigliabile utilizzare sempre delle label.** Nessun flag di stato è interessato dall'istruzione BSR.

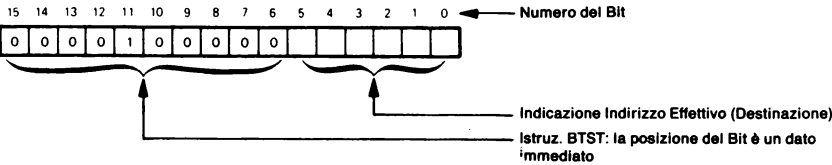
BTST (Test di un Bit)

Questa istruzione controlla un determinato bit di una locazione di memoria o di un registro dati, il cui valore viene riflesso nel flag di Zero (Z). Il bit controllato non viene modificato. Questa istruzione è, perciò, funzionalmente equivalente alla prima parte delle altre istruzioni destinate alla manipolazione di bit (BCHG, BCLR e BSET) che abbiamo descritto nelle pagine precedenti.

Esistono due versioni fondamentali. Nella prima, la posizione del bit da controllare si trova in un registro dati. Nell'altra, è specificata da un dato immediato, posto nella word successiva a quella dell'istruzione. L'operando destinazione, che contiene il bit da controllare, può essere indicato mediante uno dei seguenti tipi di indirizzamento:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Num. Bit	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento		X	111	010
Relativo al Contatore di Programma con Indice		X	111	011
Immediato	X			

Quando viene impiegato un registro dati per indicare la posizione del bit, il codice oggetto dell'istruzione BTST è:

Questo formato è sostanzialmente analogo a quello delle istruzioni per la manipolazione di bit già descritte.

Nella seconda forma dell'istruzione BTST, la posizione del bit da controllare è indicata da un dato immediato. Il relativo codice oggetto è:

Anche in questo caso, il formato è lo stesso descritto per BCHG e le altre istruzioni per il controllo dei bit. Vi rimandiamo alla descrizione dell'istruzione BCHG per una discussione più dettagliata delle modalità di esecuzione, ricordando ancora che con BTST il bit, una volta controllato, non viene modificato.

Il solo flag di stato interessato dall'istruzione BTST è il flag di Zero (Z) che è posto a uno se il bit controllato è 0, mentre se questo vale 1 viene azzerato.

CHK (Controllo che il Contenuto di un Registro sia Compreso tra Limiti Prefissati)

Questa istruzione confronta il contenuto di un registro dati con quello di un operando sorgente. Se il contenuto del registro dati è minore di zero oppure è maggiore del contenuto dell'operando sorgente, viene generata una TRAP ed il processore esegue un'appropriata routine di Exception.

L'operando sorgente, che contiene il valore da confrontare con il contenuto del registro dati, può essere indicato con uno qualsiasi dei vari modi di indirizzamento, tranne quello diretto a registro indirizzi:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto dell'istruzione CHK è:

La Figura 22-16 mostra l'esecuzione dell'istruzione CHK, con l'indirizzamento indiretto a registro indirizzi. In questa figura, il valore del registro D3 (xxxx) viene confrontato con (yyyy), che è il contenuto della locazione di memoria (wwvvvv) indicata da A3. Se xxxx è maggiore di yyyy, oppure se xxxx è minore di zero, viene prodotta una TRAP. Per la risposta all'Exception provocata da questa TRAP, il processore utilizzerà il vettore CHK, posto nella locazione di memoria 018_{16}, nella tabella dei vettori di Exception. Per una completa trattazione del modo in cui viene gestita una TRAP, vi rimandiamo al Capitolo 15 ed alla descrizione dell'istruzione TRAP contenuta nella parte successiva di questo capitolo.

Nella Figura 22-16, se xxxx è maggiore di zero e minore o uguale a yyyy, allora non si ha una TRAP e verrà eseguita l'istruzione successiva. Notate che il valore confrontato con il contenuto del registro dati è un intero in complemento a due e nel confronto sono utilizzati solo i 16 bit meno significativi del registro dati; non esiste una versione di tipo byte o long word. Il flag di Negativo (N) è posto a uno, se il contenuto del registro dati è minore di zero; verrà

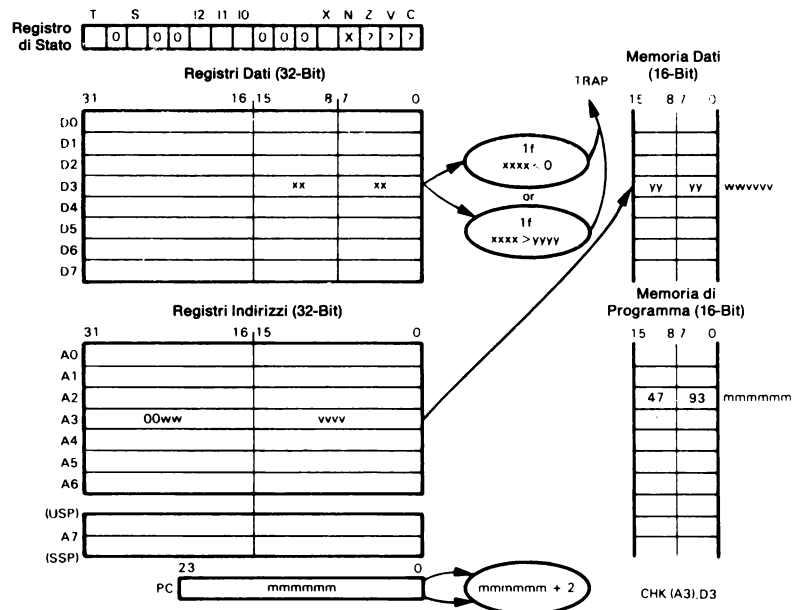



Figura 22-16. Esecuzione dell'Istruzione CHK con Indirizzamento Indiretto a registro Indirizzi.

azzerato, se nel registro dati c'è un valore superiore a quello con cui viene confrontato. I flag Z, V e C sono interessati da questa istruzione, ma in modo del tutto casuale. Il flag X non viene modificato.

Lo scopo dell'istruzione CHK è semplicemente quello di verificare se il contenuto di un registro dati si trova nell'intervallo compreso fra zero e un determinato limite superiore. Questa operazione si dimostra utile nella gestione degli array, in quanto si può porre il limite superiore, utilizzato nell'istruzione, uguale alla lunghezza dell'array (meno la lunghezza di un elemento, se l'array parte da zero). Quindi, ogni volta che si dovrà accedere all'array, basterà eseguire l'istruzione CHK per essere certi di rimanere all'interno dei limiti prefissati.

CLR (Azzeramento di un Operando)

Questa istruzione azzera un registro dati o una specificata locazione di memoria; carica, cioè, uno zero binario in quel determinato registro dati o in quella particolare locazione. I tipi di indirizzamento possibili sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione CLR è:

Come potete vedere, la grandezza del dato può essere di un byte, di una word o di una long word.

La figura 22-17 mostra l'esecuzione dell'istruzione CLR, con l'indirizzamento diretto a registro dati per indicare la destinazione. In questa figura è specificato un formato del dato pari ad una long word e, perciò, saranno azzerati tutti i 32 bit del registro D3.

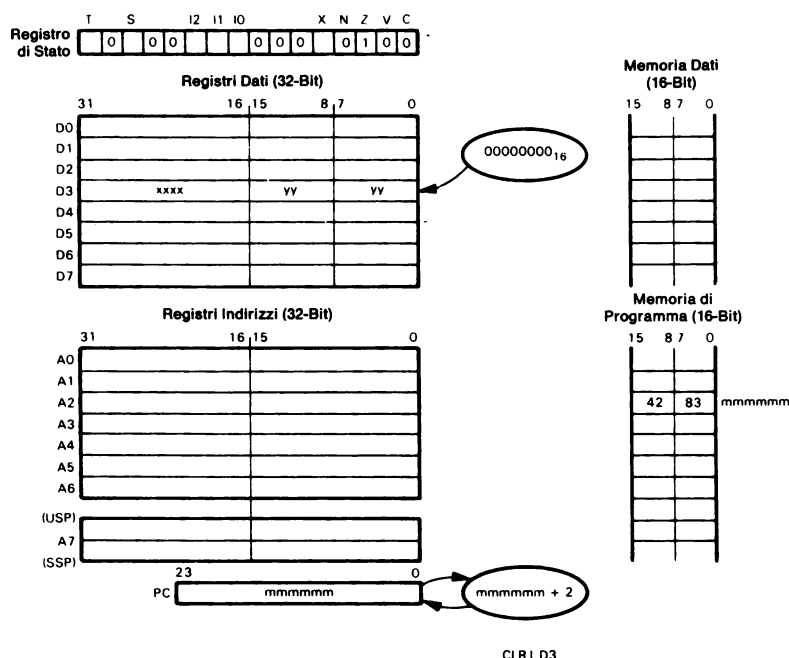



Figura 22-17. Esecuzione dell'Istruzione CLR con Indirizzamento Diretto a Registro Dati.

L'istruzione CLR pone sempre a uno il flag di Zero (Z) ed azzera sempre i flag di Negativo (N), di Overflow (V) e di Carry (C), nel registro di stato. Il flag di Extend (X) resta invariato.

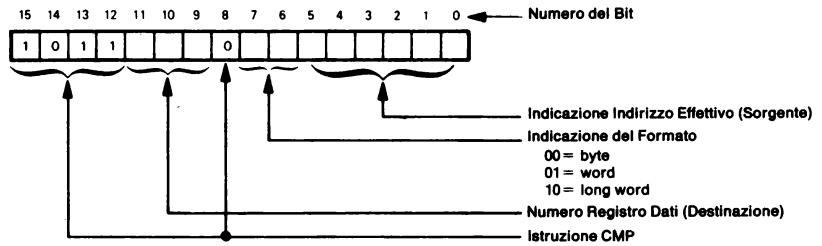
CMP (Confronto)

Questa istruzione sottrae il contenuto di un registro o di una locazione di memoria da quello di un registro dati ed in base al risultato, azzera o pone a uno i flag di stato. Nè il contenuto dell'operando sorgente, nè quello dell registro dati destinazione sono modificati. Il flag di Carry (C) rappresenta il prestito.

I tipi di indirizzamento utilizzabili per indicare l'operando sorgente sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi	X*		001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

* Non è consentito con operazioni della grandezza di un byte

Il formato del codice oggetto per l'istruzione CMP è:

La Figura 22-18 mostra l'esecuzione dell'istruzione CMP, con l'indirizzamento assoluto lungo e con D3 come registro destinazione. I flag di stato N, Z, V e C vengono tutti modificati. Il flag X resta invariato. Con gli operandi usati nella Figura 22-18, ecco ciò che accade, quando viene eseguita l'istruzione CMP $CF2000,D3:

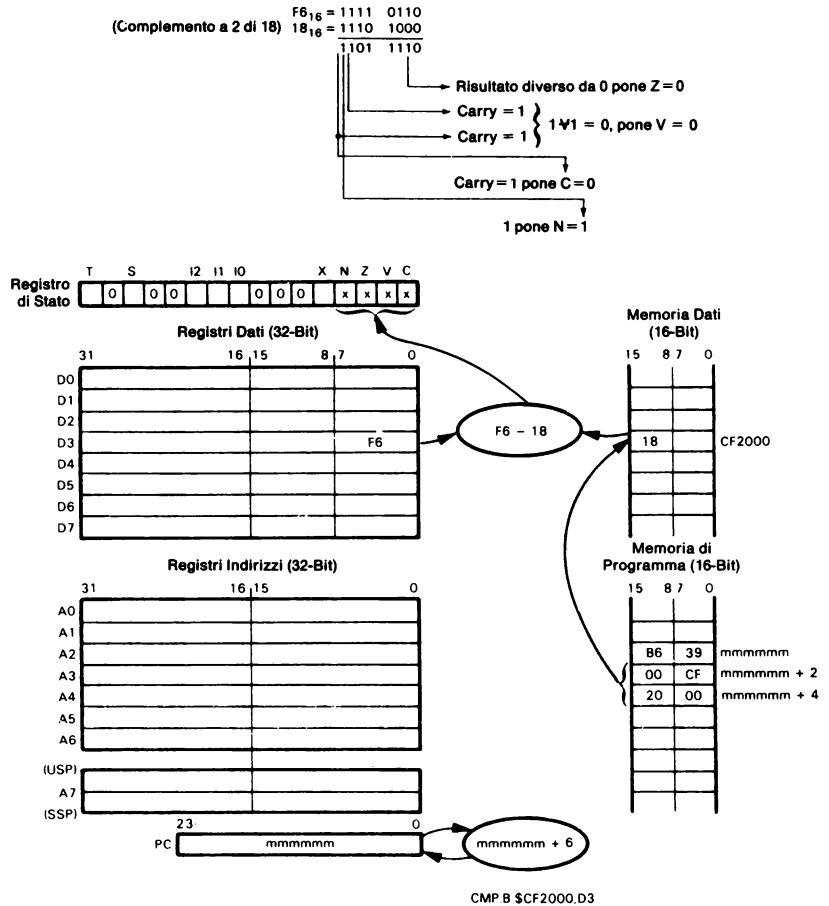



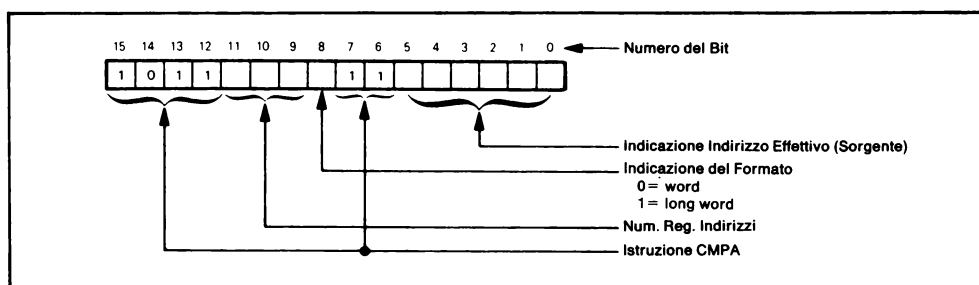
Figura 22-18. Esecuzione dell'Istruzione CMP con Indirizzamento Assoluto Lungo.

C'è il complemento del riporto risultante, dal momento che si tratta di una sottrazione ed esso rappresenta il prestito. Le istruzioni di confronto sono usate molto spesso prima delle istruzioni di salto condizionato, per assegnare ai flag gli opportuni valori.

CMPA (Confronto con un Indirizzo)

Questa istruzione è un caso speciale dell'istruzione CMP e sottrae il contenuto di un registro o di una locazione di memoria da quello di un registro indirizzi, azzerando o ponendo a uno i flag di stato conseguenza al risultato della sottrazione. Non sono modificati nè il contenuto dell'operando sorgente, nèquello del registro indirizzi destinazione. Il flag di Carry (C) rappresenta il prestito. È possibile utilizzare tutti i tipi di indirizzamento per indicare l'operando sorgente:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi	X	X	001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

L'operando destinazione deve essere un registro indirizzi.
Il codice oggetto dell'istruzione CMPA è:

Confrontando questo codice oggetto con quello dell'istruzione CMP, noterete che è identico, tranne per la grandezza del campo: la sequenza di 2 bit, che non era utilizzata con l'istruzione CMP, lo è, invece, con l'istruzione CMPA. Sono consentite soltanto delle word da 16 bit o delle long word da 32 bit, in quanto i registri indirizzi non sono in grado di gestire dati di un byte.

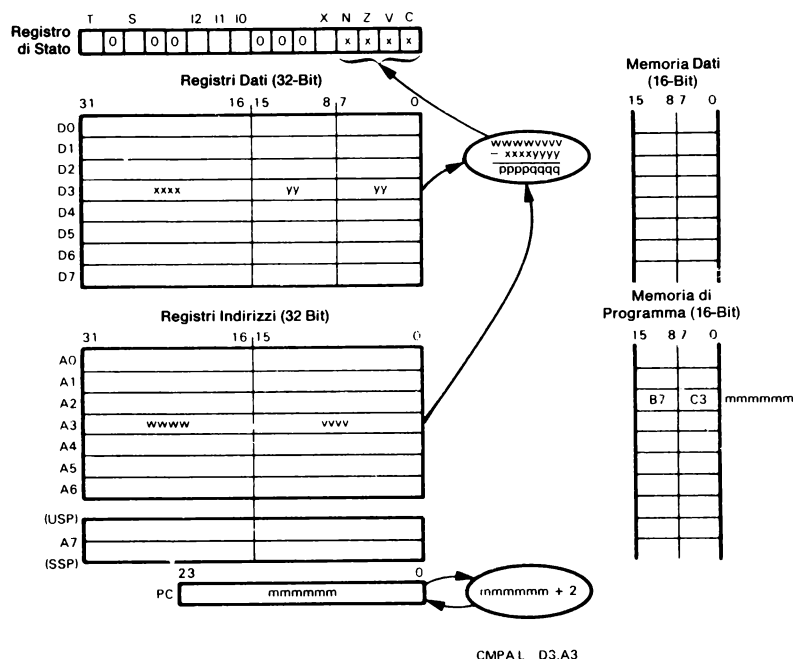



Figura 22-19. Esecuzione dell'Istruzione CMPA con Indirizzamento Diretto a Registro Dati.

La Figura 22-19 mostra l'esecuzione dell'istruzione CMPA, con il registro D3 che costituisce l'operando sorgente a 32 bit ed il registro A3 come operando destinazione.

Indicando una word da 16 bit, anzichè una long word, come operando sorgente, esso sarà trasformato in una long word mediante l'estensione del segno ed il confronto è eseguito facendo uso di tutti i 32 bit del registro indirizzi specificato.

I flag di stato sono interessati in modo analogo a quanto accade con l'istruzione CMP. Sono modificati i flag di Negativo (N), di Overflow (V), di Zero (Z) e di Carry (C), mentre il flag di Extend (X) resta invariato.

CMPI (Confronto con un Dato Immediato)

Questa istruzione sottrae il dato immediato presente nella successiva locazione della memoria di programma da un operando destinazione, azzerando o ponendo a uno i flag di stato in base al risultato della sottrazione. Il contenuto dell'operando destinazione resta immutato.

I tipi di indirizzamento utilizzabili per indicare l'operando destinazione sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			

Il codice oggetto per l'istruzione CMPI è:

La grandezza dell'operazione CMPI può essere di un byte, di una word o di una long word. Il dato immediato segue, in memoria, la word d'istruzione e deve corrispondere al formato indicato. Perciò, una o due word di dato immediato seguiranno il codice operativo dell'istruzione nella memoria di programma. Se l'istruzione indica un operando di un byte, viene usato il byte di ordine basso (secondo) della word del dato immediato. L'assemblatore provvede automaticamente a prendere il byte giusto.

La Figura 22-20 mostra l'esecuzione dell'istruzione CMPI, con un'operazione di grandezza pari ad una word (16 bit) e con l'impiego dell'indirizzamento assoluto corto. Come potete vedere, la word successiva al codice operativo dell'istruzione contiene il dato immediato (4544_{16} in questo caso). L'indirizzo assoluto corto, che subisce l'estensione del segno per indicare l'operando destinazione, segue il dato immediato. Questo verrà sottratto dal contenuto della locazione 0420_{16}.

Una volta eseguita l'istruzione CMPI #$4544,$420, i flag saranno azzerati o posti a uno nel modo indicato dallo schema seguente.

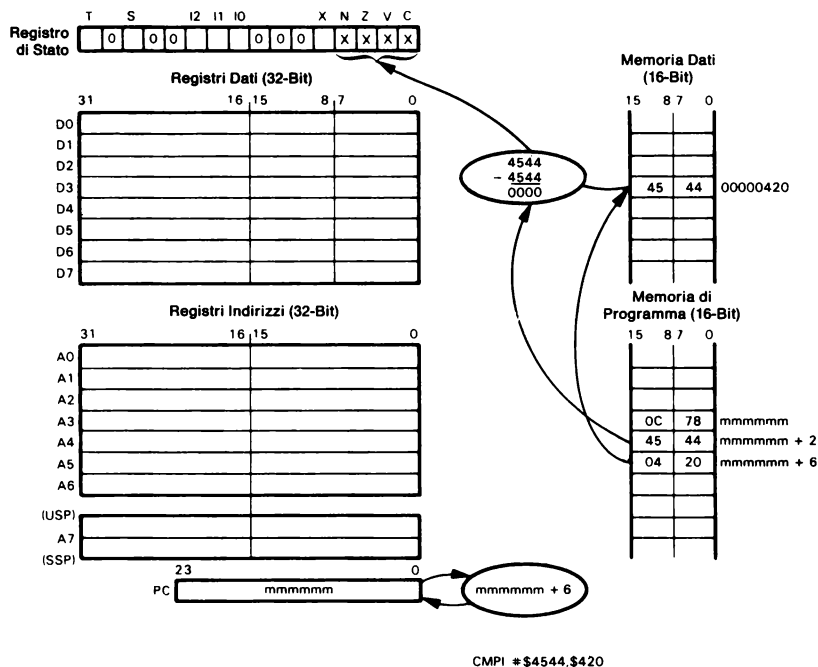
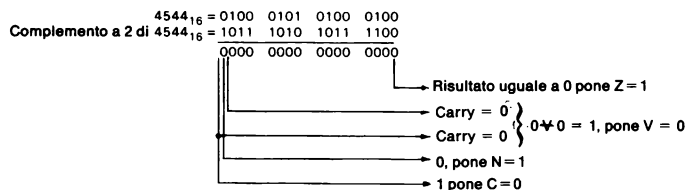



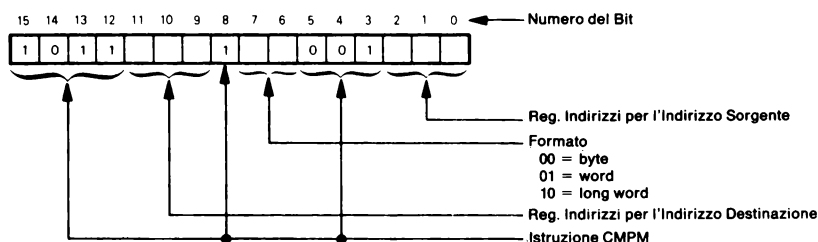
Figura 22-20. Esecuzione dell'Istruzione CMPI con Indirizzamento Assoluto Corto.

Si noti come C sia il complemento del riporto risultante, dal momento che si tratta di una sottrazione ed esso rappresenta il prestito.

Il contenuto dell'operando destinazione non varia in seguito al confronto. I flag di Negativo (N), di Overflow (V), di Zero (Z) e di Carry (C) sono posti a uno o azzerati a seconda del risultato della sottrazione, con C che viene posto a uno se si verifica un prestito. L'esecuzione dell'Istruzione non ha alcun effetto sul flag X di Extend.

CMPM (Confronto tra Valori in Memoria)

Questa istruzione confronta il contenuto di due locazioni di memoria ed azzera o pone a uno i flag di stato in base al risultato del confronto. L'operando sorgente, quello destinazione e gli indirizzi sono contenuti all'interno di registri indirizzi e viene sempre usato l'indirizzamento indiretto a registro con postincremento. Il conte-

nuto della locazione di memoria sorgente e quello della locazione di memoria destinazione non sono modificati in seguito all'operazione di confronto.

Il codice oggetto del''istruzione CMPM è:

L'istruzione CMPM può confrontare dati in memoria di grandezza pari ad un byte, una word o una long word.

La Figura 22-21 mostra l'esecuzione dell'istruzione CMPM, con il registro A4 che fornisce l'indirizzo dell'operando sorgente ed il registro A1 quello dell'operando destinazione. Il contenuto di questi due registri indirizzi viene usato per accedere agli operandi in memoria.

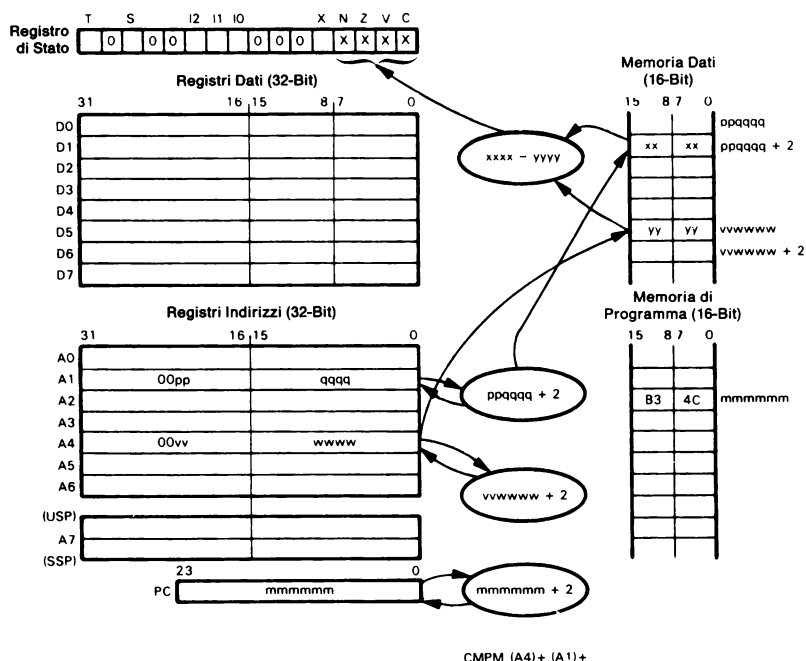



Figura 22-21. Esecuzione dell'Istruzione CMPM.

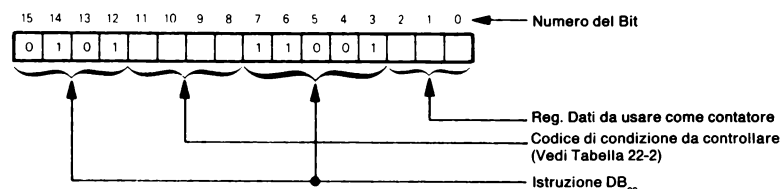
Una volta prelevati gli operandi ed effettuati i confronti, viene incrementato il contenuto di entrambi i registri indirizzi. Nella Figura 22-12 viene usato un operando della grandezza di una word e, perciò, alla fine dell'istruzione, i registri indirizzi saranno incrementati di due. Con un operando di un byte, i registri indirizzi sarebbero stati incrementati di 1, mentre con una long word l'incremento sarebbe stato di 4.

Il contenuto di entrambe le locazioni di memoria non viene modificato dall'operazione di confronto. L'istruzione CMPM interessa i flag di Negativo (N), di Overflow (V), di Zero (Z) e di Carry (C) del registro di stato. Il flag di Extend (X) resta immutato.

Questa versione dell'istruzione Compare può essere impiegata per una ricerca all'interno di una stringa. Potrebbe far parte del ciclo di istruzioni, che esegue la ricerca, a partire dall'indirizzo più basso per finire a quello più alto. Ogni volta, l'operando destinazione è confrontato con l'operando sorgente. Il microprocessore Z8000 dispone di analoghe istruzioni di confronto ed incremento, ma si tratta di versioni più sofisticate. Infatti, esse aggiornano anche un contatore, il quale modifica un bit di stato che permetterà di stabilire quando è stata confrontata l'intera stringa. L'istruzione CMPM dell'MC68000 non ha tali capacità e questa funzione dovrà essere realizzata separatamente, mediante l'istruzione DBcc (Decremento e Diramazione).

DBCC (Controllo di una Condizione, Decremento e Diramazione)

Questa istruzione controlla sia i flag di stato, che il valore contenuto in un registro dati. Prima di tutto, controlla se la condizione specificata da cc è soddisfatta. In caso affermativo, viene eseguita l'istruzione successiva. Altrimenti, il contenuto del registro dati specificato è decrementato di 1. Se quel registro dati contiene -1, verrà eseguita l'istruzione successiva.

Se la condizione non è soddisfatta ed il registro dati specificato, dopo che è stato decrementato, non contiene -1, allora il controllo viene trasferito ad un indirizzo indicato relativamente al valore attuale del contatore di programma. Lo spostamento rispetto alla locazione dell'istruzione può essere un valore compreso fra -32766 e +32769. Il codice oggetto per l'istruzione DB_{cc} è:

L'istruzione DB_{cc} consiste sempre di due word da 16 bit: la prima è la word d'istruzione, mentre la seconda è il valore di spostamento a 16 bit. I bit da 8 a 11 della word d'istruzione indicano la condizione che deve essere controllata. **La Tabella 22-2 elenca le condizioni utilizzabili con questa istruzione** ed indica quali flag di stato sono utilizzati per stabilire se il test è riuscito o meno.

La Figura 22-22 mostra l'esecuzione dell'istruzione DBNE. Se il flag Z è uguale a zero il contenuto del contatore di programma viene semplicemente incrementato di 4 e viene eseguita l'istruzione immediatamente successiva. Se il flag di Zero vale 1, i 16 bit di ordine basso del registro dati D0 sono decrementati di 1.

Tabella 22-2. Condizioni possibili con le Istruzioni DBcc.

Mnemonici (cc)	Condizione	Campo Condizione	Test
T	Vero	0000	1
F	Falso	0001	0
HI	Alto	0010	$\overline{C} \wedge \overline{Z}$
LS	Basso o Uguale	0011	$C \vee Z$
CC	Carry = 0	0100	$\overline{C}$
CS	Carry = 1	0101	C
NE	Diverso	0110	$\overline{Z}$
EQ	Uguale	0111	Z
VC	Non Overflow	1000	$\overline{V}$
VS	Overflow	1001	V
PL	Più	1010	$\overline{N}$
MI	Meno	1011	N
GE	Maggiore o Uguale	1100	$N \wedge V) \vee (\overline{N} \wedge \overline{V})$
LT	Minore	1101	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
GT	Maggiore	1110	$(N \wedge V \wedge \overline{Z}) \vee (\overline{N} \wedge \overline{V} \wedge \overline{Z})$
LE	Minore o Uguale	1111	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$

Figura 22-22. Esecuzione dell'Istruzione DBNE.

Se i 16 bit di ordine basso del registro di D0, dopo essere stati decrementati, contengono -1, allora il contatore di programma viene semplicemente decrementato di 4 ed è eseguita l'istruzione immediatamente successiva.

Se il flag di Zero (Z) è uguale a 0 ed il risultato ottenuto decrementando il contenuto del registro D0 non è = -1, significa che il loop non è ancora completo e viene, perciò, effettuata una diramazione, sommando il valore di spostamento a 16 bit, con estensione del segno, al valore del contatore di programma, dopo che questo è stato incrementato di due. Perciò, **la logica di questa diramazione è il contrario di quella dell'istruzione di diramazione condizionata (B_{cc}):** con B_{cc} il valore di spostamento era sommato al contatore di programma, quando era soddisfatta la condizione specificata (cc), mentre, **nel caso di DB_{cc}, la diramazione avviene se la condizione (cc) non è soddisfatta e se non si è ancora verificato un numero sufficiente di iterazioni all'interno del ciclo.**

L'istruzione DB_{cc} facilita la realizzazione di cicli ripetitivi, dal momento che essa controlla il registro dei codici di condizione e fornisce un contatore del ciclo in uno dei registri dati. Per una trattazione completa di questa istruzione e delle relative modalità di impiego vi rimandiamo al Capitolo 6.

Non viene modificato nessuno dei flag di stato.

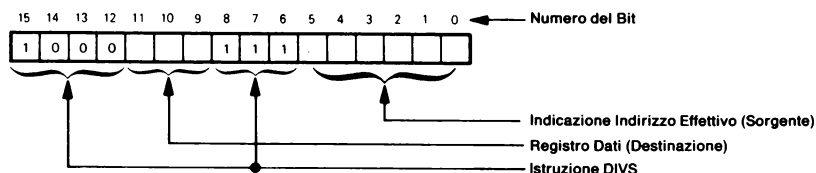
DIVS (Divisione con Segno)

Questa istruzione divide un operando a 32 bit, contenuto nel registro dati destinazione, con il valore a 16 bit contenuto nell'operando sorgente. L'operazione viene eseguita usando l'aritmetica binaria in complemento a due. Si ottiene un risultato a 32 bit nel registro dati destinazione.

L'operando sorgente può essere indicato con uno qualsiasi dei vari tipi di indirizzamento, tranne quello diretto a registro indirizzi:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il risultato a 32 bit ottenuto con l'istruzione DIVS è formato dal quoziente, che occupa i 16 bit meno significativi del registro dati destinazione, e dal resto, contenuto nei 16 bit più significativi. Il resto ha lo stesso segno del dividendo, tranne quando è zero.

Il codice oggetto dell'istruzione DIVS è:

La Figura 22-23 mostra l'esecuzione dell'istruzione DIVS, con l'indirizzamento indiretto a registro indirizzi per indicare l'operando sorgente.

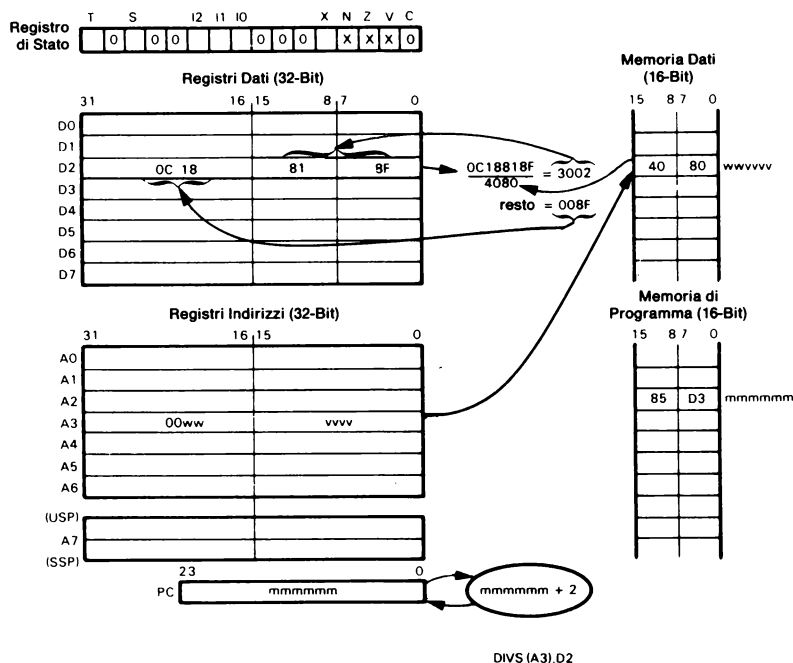



Figura 22-23. Esecuzione dell'Istruzione DIVS con Indirizzamento Indiretto a Registro Indirizzi.

Dopo che è stata eseguita l'istruzione mostrata nella figura, il registro D2 contiene il resto ($008F_{16}$) nei 16 bit più significativi ed il quoziente (3002_{16}) nei 16 bit meno significativi.

Il flag di Negativo (N) del registro di stato è posto a uno se il quoziente è negativo; altrimenti viene azzerato. Se si verifica un Overflow lo stato del flag N è indefinito. Il flag di Zero (Z) è posto a uno se il quoziente è zero, altrimenti viene azzerato. Anche lo stato di questo flag è indefinito nel caso di un overflow. Si verifica un

overflow se l'operando sorgente è maggiore dell'operando destinazione. Questo viene rilevato prima che abbia luogo la divisione: il flag di Overflow (V) è posto a uno e gli operandi resteranno invariati. Il flag di Carry (C) è sempre azzerato. Il flag di Extend (X) resta immutato.

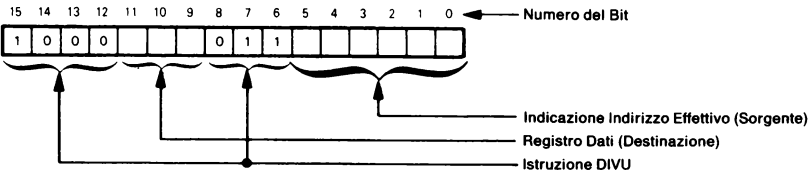
Se viene tentata una divisione per zero l'istruzione non è eseguita, si verifica una TRAP ed il processore esegue l'appropriata routine di Exception. Il vettore di Exception utilizzato è quello della locazione 014_{16} (vettore #5), corrispondente appunto alle Trap dovute ad una divisione per zero. Per una descrizione della sequenza di eventi che hanno luogo in un caso come questo, vi rimandiamo all'istruzione TRAP ed alla trattazione delle Exception, nel Capitolo 15.

DIVU (Divisione senza Segno)

Questa istruzione divide un operando a 32 bit, contenuto nel registro dati destinazione, con un operando sorgente a 16 bit. La divisione viene eseguita usando l'aritmetica binaria senza segno. Si ottiene un risultato a 32 bit, nel registro dati destinazione. L'operando sorgente può essere specificato usando uno qualsiasi dei vari tipi di indirizzamento, tranne quello diretto a registro indirizzi:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il risultato a 32 bit ottenuto con l'istruzione DIVU è memorizzato nel registro destinazione, con il resto che occupa i 16 bit più significativi ed il quoziente i 16 bit meno significativi.

Il codice oggetto per l'istruzione DIVU è:

La Figura 22-24 mostra l'esecuzione dell'istruzione DIVU, con l'uso dell'indirizzamento indiretto a registro indirizzi per indicare l'operando sorgente. Con gli operandi usati nell'esempio della figura una volta eseguita l'istruzione DIVU il resto ($004C_{16}$) sarà contenuto nei 16 bit più significativi del registro D2 ed il quoziente ($5D25_{16}$) nei 16 bit meno significativi, sempre del registro D2.

Il flag di Negativo (N) diventa uno se il bit più significativo del quoziente è 1, altrimenti viene azzerato. In caso di overfolw il valore del flag N resta indefinito. Gli altri flag di stato sono interessati in modo analogo a quanto avviene con l'istruzione DIV.

Se si cerca di eseguire una divisione per zero, viene generata una TRAP ed il processore dà inizio, automaticamente, all'appropriata routine di Exception. Il vettore di Exception utilizzato è quello nella locazione 014_{16} (vettore #5), assegnato alle Trap provocate da una divisione per zero. Per una descrizione della sequenza di eventi che hanno luogo in un caso come questo, vi rimandiamo all'istruzione TRAP ed alla trattazione delle Exception, nel Capitolo 15.

EOR (OR Logico Esclusivo)

Questa istruzione esegue un OR esclusivo, bit a bit, del contenuto di un registro dati con il contenuto dell'operando destinazione e salva il risultato sostituendolo all'operando destinazione. Quest'ultimo può essere indicato mediante uno dei tipi di indirizzamento mostrati nella tabella.

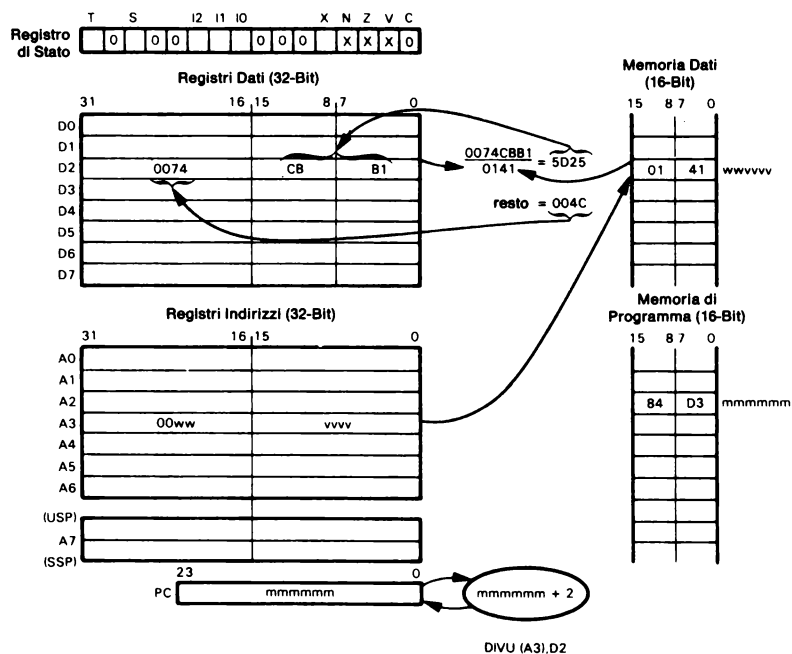



Figura 22-24. Esecuzione dell'Istruzione DIVU con Indirizzamento Indiretto a registro Indirizzi.

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice ogeetto per l'istruzione EOR è:

Figura 22-25. Esecuzione dell'Istruzione EOR con Indirizzamento Indiretto a Registro Indirizzi.

La Figura 22-25 mostra l'esecuzione dell'istruzione EOR con l'indirizzamento indiretto a registro indirizzi. Usando come operandi i valori mostrati in questa figura, dopo che è stata eseguita l'istruzione EOR.B, gli 8 bit meno significativi del registro D3 conterranno 43_{16}.

L'OR esclusivo logico è analogo ad un'operazione "non uguale" eseguita bit a bit; cioè, l'output è 1 se e solo se gli input non sono uguali. **EOR è usato per rilevare eventuali modifiche dei bit di stato e per il calcolo della parità e di altri codici destinati ad individuare e correggere eventuali errori.**

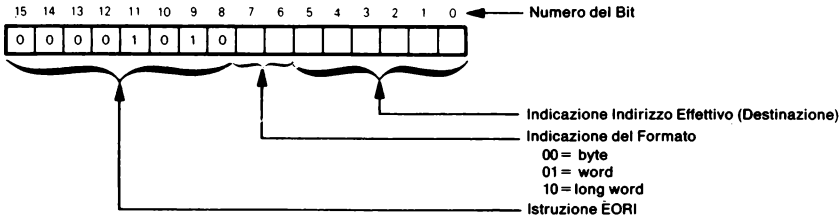
Il flag di negativo (N) è posto a uno, se il bit più significativo del risultato è 1; altrimenti viene azzerato. Il flag di Zero (Z) diventa uno se il risultato è 0, altrimenti è azzerato. I flag di Overflow (V) e di Carry (C) sono sempre azzerati. Il flag di Extend (X) resta invariato.

EORI (OR Esclusivo Immediato)

Questa istruzione esegue un OR esclusivo, bit per bit, del dato immediato presente nella successiva locazione della memoria di programma, con l'operando destinazione, al posto del quale viene poi salvato il risultato. Per indicare la locazione di destinazione si può utilizzare uno dei seguenti tipi di indirizzamento.

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			
Registro di Stato		X	111	100

L'operando destinazione può essere costituito anche dai codici di condizione o dall'intero registro di stato. **Se la destinazione è l'intero registro di stato si ha un'istruzione privilegiata, che potrà essere eseguita solo quando il processore è nel modo Supervisore.**

Il codice oggetto dell'istruzione EORI è:

Il formato dell'operazione EORI può essere di un byte, di una word o di una long word. Il dato immediato segue la word d'istruzione e deve corrispondere alla grandezza specificata. Perciò, il codice operativo sarà seguito da una o due word di dato immediato. Se è specificato un operando di un byte, viene utlizzato il byte di ordine basso (secondo) della word del dato. È l'assemblatore che preleva, automaticamente, il byte giusto. Se l'istruzione si riferisce al registro di stato e l'operazione è di un byte, l'operando destinazione è il byte di ordine basso del registro di stato, quello contenente i codici di condizione. Se l'operazione è di una word, l'operando destinazione sarà l'intero registro di stato e si tratterà di un'istruzione privilegiata.

La Figura 22-26 mostra l'esecuzione dell'istruzione EORI con un operando di una word (16 bit) e facendo uso dell'indirizzamento assoluto corto. Come potete vedere, la word successiva al codice operativo dell'istruzione contiene il dato immediato ($B31C_{16}$ in questo esempio). L'indirizzo assoluto corto, che subisce l'estensione del segno per indicare l'operando destinazione, segue il dato immediato nella memoria di programma. Viene eseguito un OR esclusivo del dato immediato con il contenuto della locazione di memoria 420_{16}.

I flag di Negativo (N) e di Zero (Z) sono modificati dall'istruzione EORI in base al risultato ottenuto. I flag di Overflow (V) e di Carry (C) sono, invece, sempre azzerati. Il flag di Extend (X) resta immutato. Naturalmente, se l'operando destinazione è costituito dai codici di condizione, o dall'intero registro di stato, allora tutti i codici di condizione saranno modificati in seguito all'esecuzione della stessa istruzione.

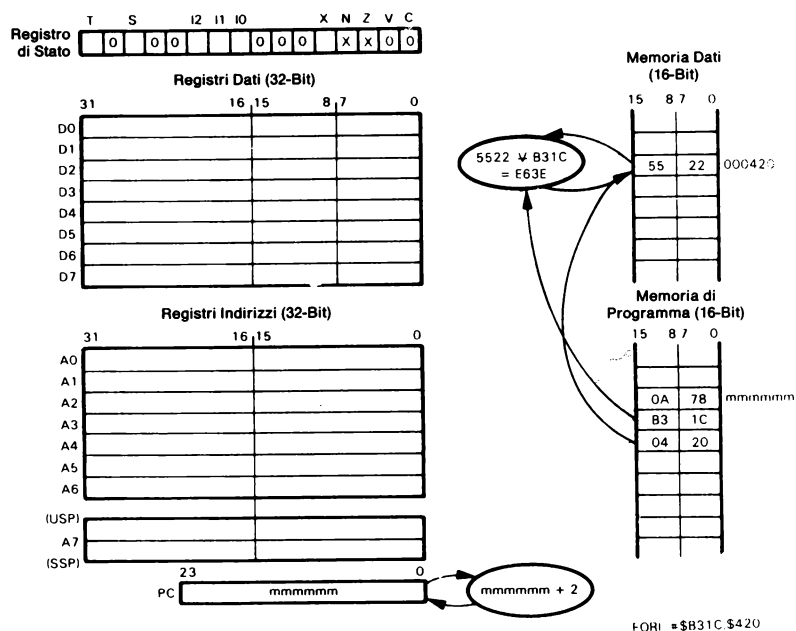



Figura 22-26. Esecuzione dell'Istruzione EORI con Indirizzamento Assoluto Corto.

EXG (Scambio del Contenuto di Due Registri)

Questa istruzione scambia il contenuto di due registri a 32 bit. Può trattarsi di due registri dati, due registri indirizzi o di un registro dati e uno indirizzi.
Sono possibili soltanto indirizzamenti diretti a registro:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi	X	X	001	rrr
Indiretto a Registro Indirizzi				
Indiretto a Registro con Postincremento				
Indiretto a Registro con Predecremento				
Indiretto a Registro con Spostamento				
Indiretto a Registro con Indice				
Assoluto Corto				
Assoluto Lungo				
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto per l'istruzione EXG è:

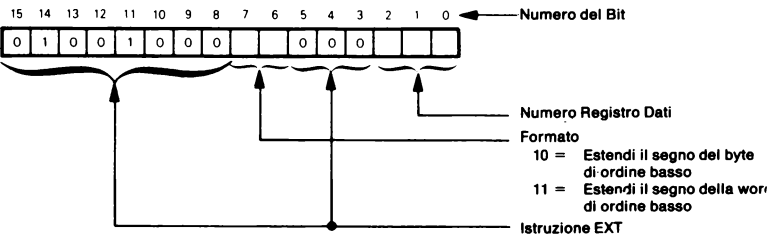
La Figura 22-27 mostra l'esecuzione dell'istruzione EXG, con lo scambio del contenuto dei registri D2 e A1. Viene scambiato sempre l'intero valore a 32 bit di entrambi i registri.
Naturalmente, EXG è simmetrica: ad es., EXG A1, D2 e EXG D2,A1 sono la stessa operazione.
Non viene interessato nessuno dei flag di stato.

EXT (Estensione del Bit di Segno)

Questa istruzione fa sì che il bit di segno (bit 7) di un byte sia esteso ai bit 8-15 della word di un registro dati oppure che il bit di

Figura 22-27. Esecuzione dell'Istruzione EXG.

Figura 22-28. Esecuzione dell'Istruzione EXT.

segno di una word (bit 15) sia esteso ai bit 16-32 di un registro dati. Il codice oggetto di un'istruzione EXT è:

La Figura 22-28 mostra l'esecuzione dell'istruzione EXT, con un'operando di una word. Dal momento che il bit 7 del registro D5 è 0, anche i bit da 8 a 15 saranno posti a 0 dall'esecuzione dell'istruzione EXT. Se viene specificato che la grandezza dell'operazione deve essere di una long word, saranno i bit da 16 a 31 che assumeranno il valore del bit 15. Perciò, se l'istruzione mostrata nella Figura 22-28 fosse seguita da un'istruzione EXT.L D5, anche i bit da 16 a 31 del registro D5 diventerebbero 0.

Il flag di Negativo (N) diventa uno se il risultato dell'operazione EXT genera un numero negativo, altrmenti viene azzerato. Il flag di Zero (Z) è posto a uno, se il risultato è 0, in caso contrario viene azzerato. I flag di Overflow (V) e di Carry (C) sono sempre azzerati. Il flag di Extend (X) resta invariato.

JMP (Salto)

Questa istruzione provoca un salto incondizionato all'indirizzo di memoria specificato. I tipi di indirizzamento possibili sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento				
Indiretto a Registro con Predecremento				
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento		X	111	010
Relativo al Contatore di Programma con Indice		X	111	011
Immediato				

Il codice oggetto dell'istruzione JMP è:

La Figura 22-29 mostra l'esecuzione dell'istruzione JMP, con l'indirizzamento indiretto a registro. Quando è eseguita l'istruzione JMP (A5), nel contatore di programma viene caricato l'indirizzo contenuto nelle locazioni di memoria ppqqqq e ppqqqq+2.

Nessuno dei flag di stato viene modificato da questa istruzione. Il precedente valore del contatore di programma va perso.

JSR (Salto ad una Subroutine)

Questa istruzione provoca un salto incondizionato all'indirizzo di memoria specificato, salvando il vecchio valore del contatore di programma allo stack di sistema. I modi di indirizzamento utilizzabili con l'istruzione JSR sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento				
Indiretto a Registro con Predecremento				
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento		X	111	010
Relativo al Contatore di Programma con Indice		X	111	011
Immediato				

Figura 22-29. Esecuzione dell'Istruzione JMP con Indirizzamento Indiretto a registro Indirizzi.

Il codice oggetto per l'istruzione JSR è:

La Figura 22-30 mostra l'esecuzione dell'istruzione JSR, con l'impiego dell'indirizzamento assoluto corto. Dopo che è stata eseguita l'istruzione illustrata nella Figura 22-30 l'esecuzione del programma continuerà a partire dalla locazione di memoria 000400_{16}. Il valore del contatore di programma alla fine dell'istruzione JSR (mmmmmm+4), sarà stato salvato allo stack Utente. Il valore finale del puntatore allo stack Utente sarà inferiore di 4 rispetto a quello iniziale.

JSR è analoga a JMP, tranne per il fatto che JSR salva allo stack il vecchio valore del contatore di programma, fornendo, così, un collegamento con la subroutine. Un'istruzione RTS, alla fine della subroutine, restituisce il controllo all'istruzione immediatamente successiva a JSR, ammesso che la subroutine non abbia modificato l'indirizzo di ritorno, nè il puntatore allo stack. JSR provoca un salto assoluto e incondizionato ad una subroutine, a differenza di BSR che determina, invece, un salto relativo. Tuttavia, l'istruzione JSR *+300 ha lo stesso effetto di BSR *+300.

Nessuno dei flag di stato viene modificato.

Questa istruzione forma un indirizzo effettivo utilizzando uno dei vari tipi di indirizzamento e carica l'indirizzo ottenuto nel registro indirizzi specificato. I modi di indirizzamento possibili sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati				
Diretto a Registro Indirizzi		X		
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento				
Indiretto a Registro con Predecremento				
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato				

Figura 22-30. Esecuzione dell'Istruzione JSR con Indirizzamento Assoluto Corto.

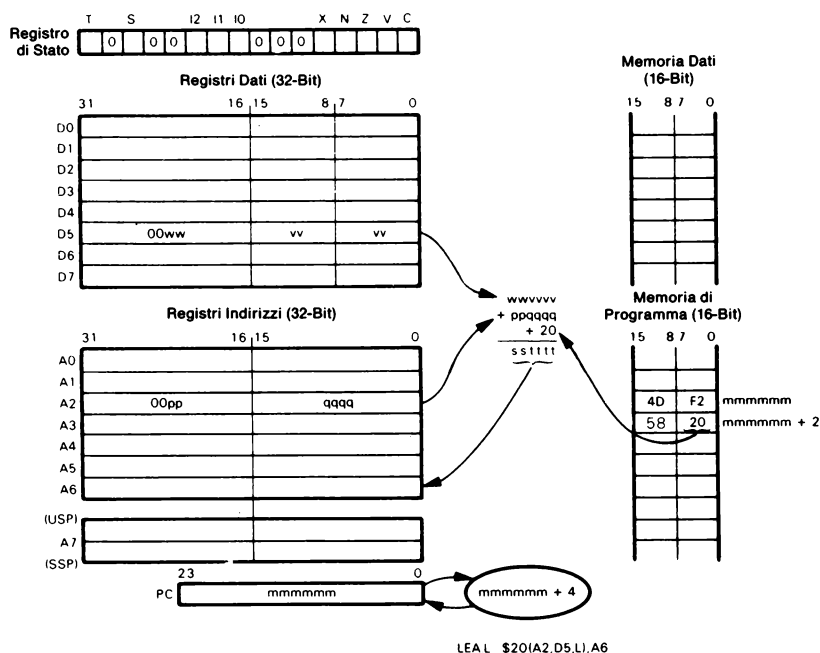
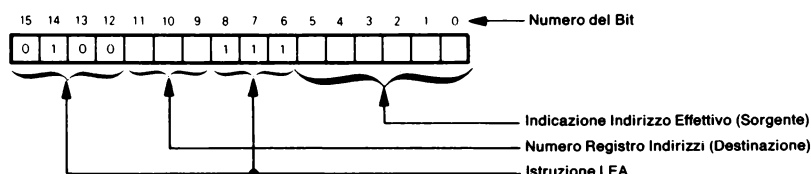



Figura 22-31. Esecuzione dell'Istruzione LEA con Indirizzamento Indiretto a Registro Indirizzi con Indice Lungo.

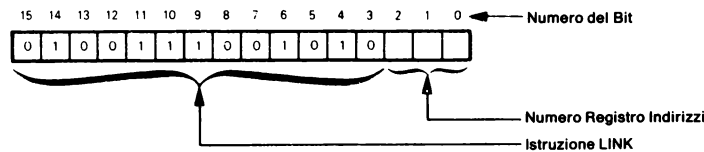
Il codice oggetto dell'istruzione LEA è:

La Figura 22-31 mostra l'esecuzione dell'istruzione LEA, con l'impiego dell'indirizzamento indiretto a registro indirizzi con indice. In questa figura, il contenuto del registro D5 viene sommato a quello del registro A2 e, quindi, a questi due valori a 32 bit viene aggiunto il valore di spostamento (20_{16}), posto dopo la word d'istruzione. L'indirizzo a 32 bit (sstttt) ottenuto viene caricato nel registro A6.

Come potete vedere, una volta eseguita l'istruzione LEA, è l'indirizzo ottenuto che viene messo nel registro destinazione e non il contenuto della word di memoria che esso indica.

Non è interessato nessuno dei flag di stato.

LINK (Link e Allocazione)

Questa istruzione mette sullo stack di Sistema il contenuto di un determinato registro indirizzi. Il contenuto aggiornato del puntato-

re allo stack è caricato nel registro indirizzi specificato. Infine, al puntatore allo stack viene aggiunto un offset in complemento a due, presente nella word successiva a quella dell'istruzione, con estensione del bit di segno. Il codice oggetto dell'istruzione LINK è:

La Figura 22-32 mostra l'esecuzione dell'istruzione LINK.

L'istruzione LINK utilizza il puntatore allo stack, uno degli altri registri indirizzi (come puntatore di "frame") ed un valore di spostamento. È utilizzata soprattutto all'inizio di una subroutine. Per prima cosa, mette il valore attuale del puntatore di frame sullo stack di Sistema. Il valore del puntatore allo stack, opportunamente decrementato, è caricato sul puntatore di frame, il quale indica così la nuova sommità allo stack, che diventerà anche la sommità della frame.

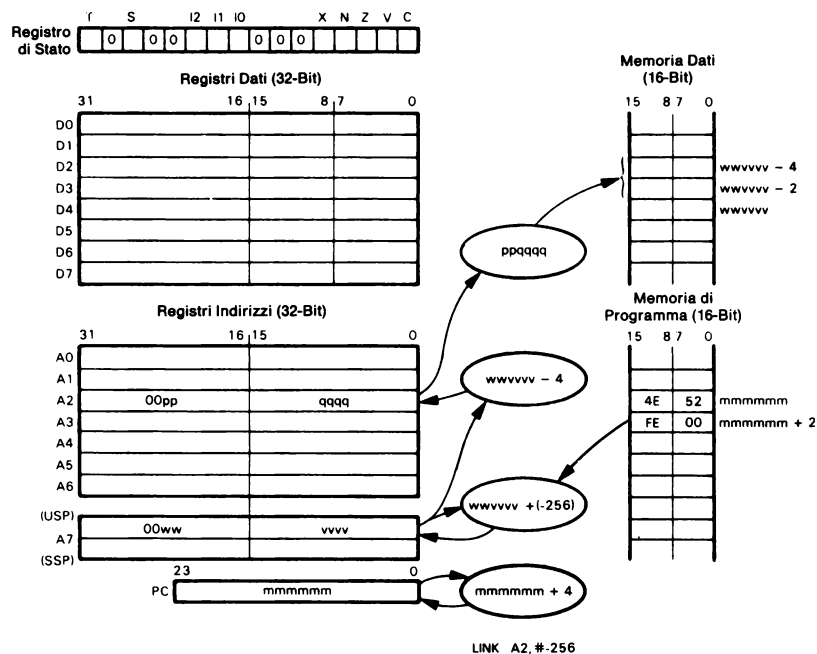
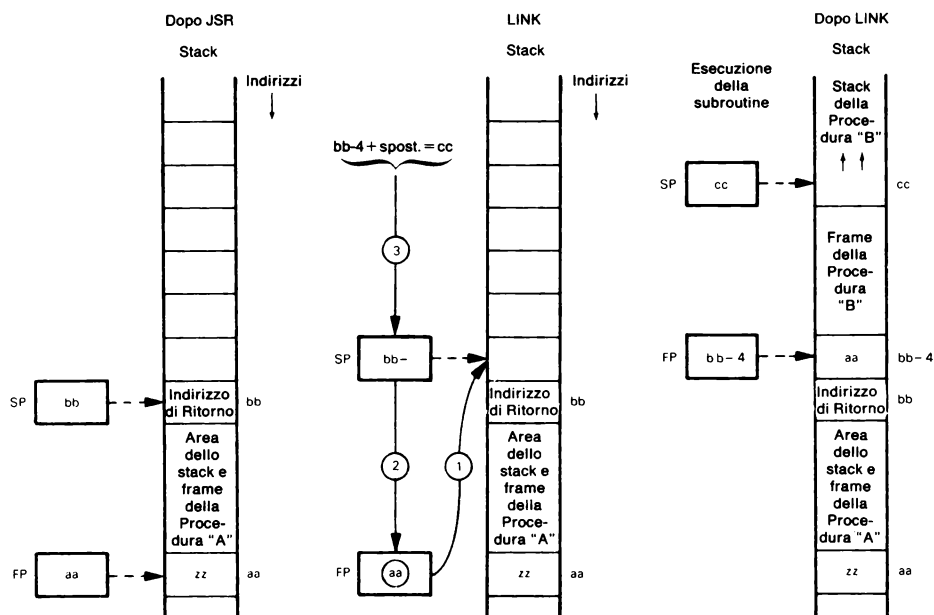
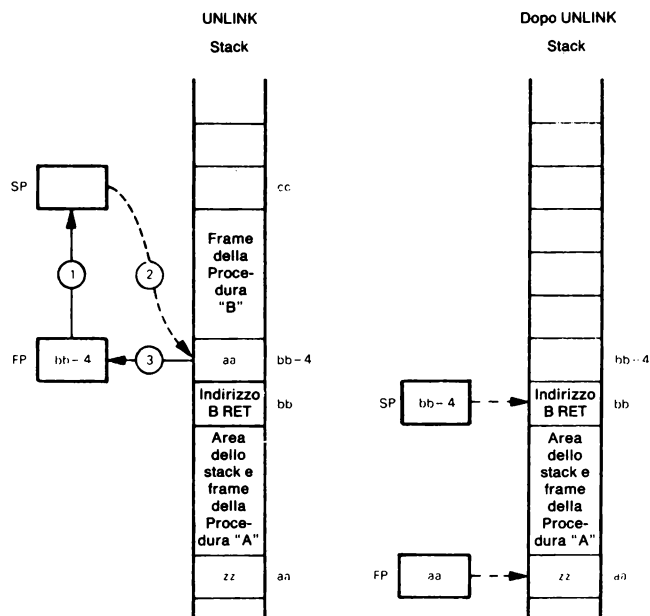



Figura 22-32. Esecuzione dell'Istruzione LINK.

Infine, lo spostamento fornito con l'istruzione LINK viene usato per decrementare il puntatore allo stack, in modo da liberare un'area ("frame") di memoria per mettervi variabili locali e parametri. La subroutine potrà, quindi, accedere a queste variabili attraverso il

puntatore di frame. Al termine della subroutine, viene usata l'istruzione UNLK (Unlink) per ripulire lo stack.

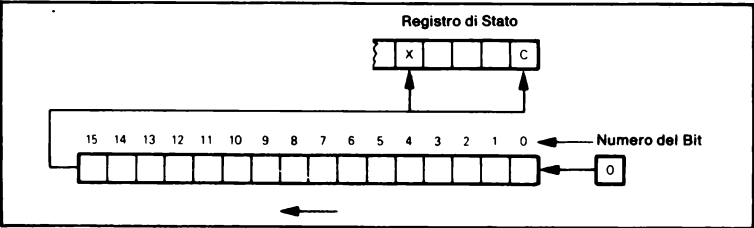
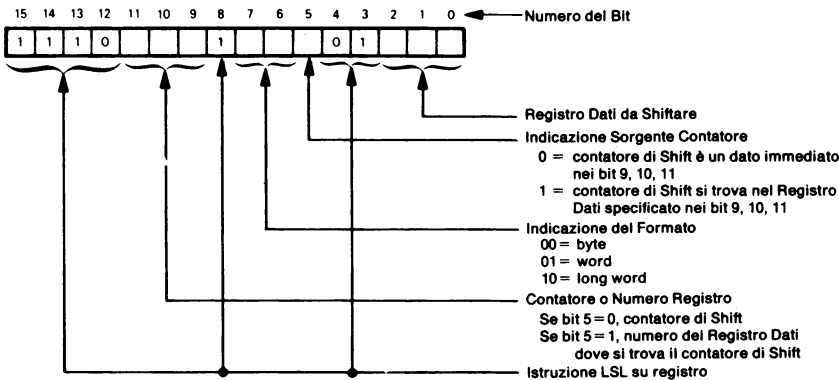
Facciamo un esempio. Nello schema seguente il puntatore allo stack, dopo che la procedura A ha eseguito una JSR alla procedura B, conterrà l'indirizzo bb. Il puntatore allo stack indica la locazione dello stack dove si trova l'indirizzo della procedura A, cui la procedura B deve ritornare. A questo punto, il puntatore di frame contiene l'indirizzo aa, che indica la sommità della frame della procedura A. Quando viene eseguita l'istruzione LINK, ecco ciò che accade: Per prima cosa, (1) l'indirizzo aa, presente nel puntatore di frame, viene messo allo sullo tramite indirizzamento con predecremento, andando ad occupare la locazione bb-4. Quindi, (2) il contenuto di SP, in questo caso bb-4, viene caricato sul puntatore di frame (FP). Infine, (3) il valore di spostamento, fornito con l'istruzione LINK, è sommato al contenuto del puntatore allo stack. Perciò, una volta eseguita l'istruzione LINK, SP conterrà un nuovo valore (cc nel nostro caso) ed il puntatore di frame conterrà l'indirizzo bb-4. In tal modo, sullo stack viene riservata un'area, di grandezza uguale al valore di spostamento fornito con l'istruzione LINK e destinata a contenere le variabili locali della procedura B.

Al termine della procedura B che ha eseguito l'istruzione LINK, è necessaria un'istruzione UNLK per ripristinare le condizioni preesistenti. L'illustrazione successiva mostra la sequenza degli eventi che si verificano durante l'istruzione UNLK.

Innanzitutto, (1) il contenuto del puntatore di frame viene caricato sul puntatore allo stack. Quindi, (2) il puntatore allo stack viene usato per accedere allo stack e caricare sul puntatore di frame, mediante indirizzamento con postincremento, il contenuto di quella determinata locazione dello stack. Confrontando questa illustrazione con quella precedente, potete constatare come, dopo l'esecuzione dell'istruzione UNLK, siano state ripristinate le condizioni precedenti all'istruzione LINK: il puntatore di frame (FP) contiene l'indirizzo aa ed il puntatore allo stack (SP) contiene l'indirizzo bb, come era inizialmente. Quando viene eseguita la prevista istruzione RTS (Ritorno da una Subroutine) verrà prelevato dalla locazione bb dello stack l'indirizzo di ritorno, mediante indirizzamento con postincremento.

L'istruzione LINK non modifica nessuno dei flag di stato.

Essa consente di utilizzare un valore di spostamento a 16 bit, che, prima di essere sommato al puntatore allo stack, subisce l'estensione del segno. **Quando vi servite dell'istruzione LINK per creare delle frame sullo stack, assicuratevi di utilizzare un valore di spostamento *negativo*.** Un valore positivo finirebbe per riscrivere su di un'area già utilizzata, distruggendo le variabili in essa contenute.

LSL (Shift Logico a Sinistra in un Registro Dati)

Questa istruzione effettua lo shift a sinistra del contenuto di un determinato registro dati. È analoga all'istruzione ASL (Shift Aritmetico a Sinistra), cui vi rimandiamo per una descrizione dettagliata

delle modalità di esecuzione. Lo shift viene eseguito nel modo seguente:

Il codice oggetto dell'istruzione LSL è:

LSL (Shift Logico a Sinistra in Memoria)

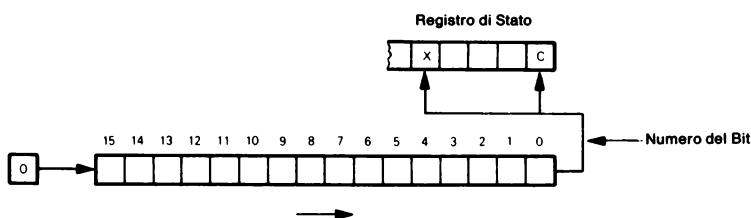
Questa istruzione esegue uno shift logico a sinistra di una posizione di una determinata locazione di memoria. I tipi di indirizzamento utilizzabili per indicare la locazione di memoria sono:

Modi di Indirizzamento Possibili	Operando Sorgente	Operando Destinazione	Campo Ind.Eff. Destinazione Modo	Campo Ind.Eff. Destinazione Num. Registro
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

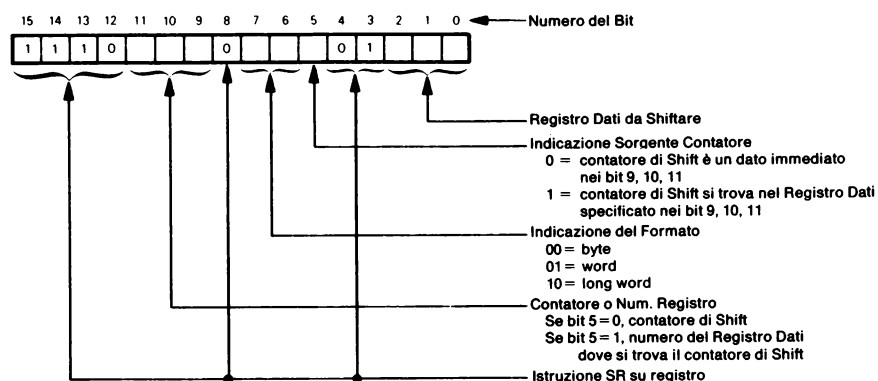
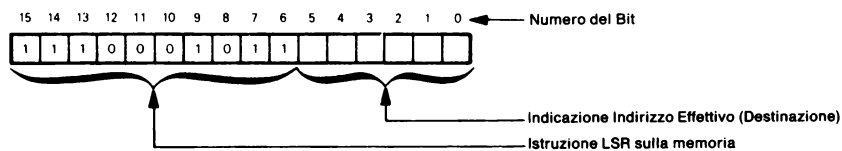
Il codice oggetto di questa istruzione è:

Questa versione dell'istruzione LSL è del tutto analoga all'istruzione ASL (Shift Aritmetico a Sinistra), cui vi rimandiamo per una descrizione dettagliata delle modaità di esecuzione. Lo shift viene eseguito nel modo descritto per l'istruzione precedente.

LSR (Shift Logico a Destra in un Registro Dati o in Memoria)

L'istruzione LSR esegue uno shift logico a destra dell'operando. È identica all'istruzione ASR, tranne per il fatto di caricare uno zero nel bit più significativo, invece di conservarlo invariato.

I flag di stato sono modificati in modo analogo a quanto accade con l'istruzione ASR. Naturalmente, nel caso dell'istruzione LSR, il flag di negativo (N) viene azzerato dal momento che il bit più significativo è posto a zero dall'esecuzione dell'istruzione. Consultate l'istruzione ASR per maggiori dettagli.

I tipi di indirizzamento utilizzabili con l'istruzione LSR sono identici a quelli descritti per l'istruzione LSL.

Il codice oggetto per la versione su registro è il seguente:

Il codice oggetto per la versione sulla memoria è:

MOVE (Trasferimento di Dati dalla Sorgente alla Destinazione)

Questa istruzione può essere utilizzata per spostare un dato dai registri alla memoria, dalla memoria ai registri o da un registro all'altro. È perciò equivalente alle istruzioni LOAD e STORE degli altri microprocessori. Esistono poche limitazioni riguardo alle modalità necessarie per indicare gli operandi sorgente e destinazione dell'istruzione MOVE:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi	X*		001	rrr
Indiretto a Registro Indirizzi	X	X	010	rrr
Indiretto a Registro con Postincremento	X	X	011	rrr
Indiretto a Registro con Predecremento	X	X	100	rrr
Indiretto a Registro con Spostamento	X	X	101	rrr
Indiretto a Registro con Indice	X	X	110	rrr
Assoluto Corto	X	X	111	000
Assoluto Lungo	X	X	111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

* Non è consentito con operazioni della grandezza di un byte

Il codice oggetto dell'istruzione MOVE è:

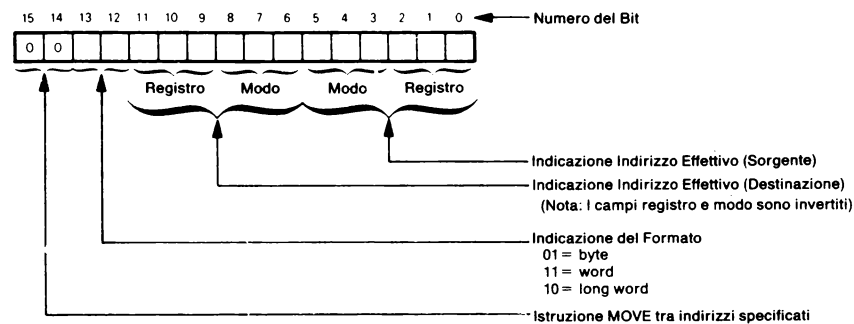
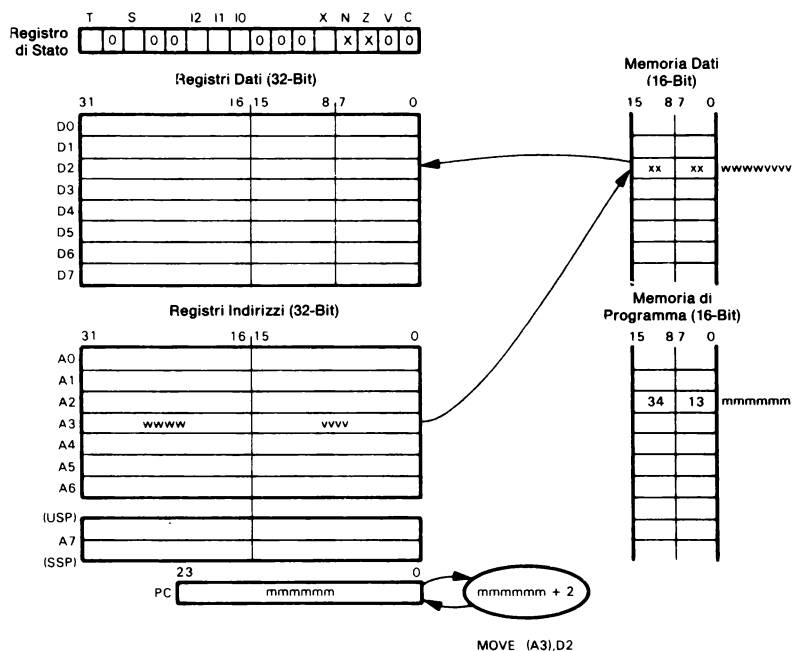



Figura 22-33. Esecuzione dell'Istruzione MOVE con Indirizzamento Indiretto a Registro Indirizzi.

Si noti come nell'indicazione dell'indirizzo di destinazione effettivo il numero del registro ed il campo indicante il modo siano disposti in modo inverso. Naturalmente, l'assemblatore provvederà automaticamente ad interpretarli correttamente.

La Figura 22-33 mostra l'esecuzione dell'istruzione MOVE, con l'indirizzamento indiretto a registro indirizzi per indicare l'operando sorgente e diretto a registro dati per l'operando destinazione. In questa figura, il registro A3 contiene l'indirizzo di memoria del dato, di lunghezza pari ad una word, che deve essere trasferito nel registro D2. Dopo che l'istruzione è stata eseguita, i 16 bit meno significativi del registro D2 conterranno il valore xxxx.

Il trasferimento dei dati in senso opposto lo possiamo ottenere mediante l'istruzione MOVE D2,(A3).

Al momento di trasferirlo, il processore esamina il dato ed, in base a questo, sono posti a uno o azzerati i flag di Zero (Z) e Negativo (N). I flag di Overflow (V) e di Carry (C) sono azzerati in ogni caso . Il flag di Extend (X) resta invariato.

MOVE in CCR (Trasferimento nei Codici Condizione)

Questa è una forma speciale dell'istruzione MOVE, che mette il contenuto dell'operando sorgente nella parte del registro di stato (il byte di ordine basso) riservato ai codici di condizione. I modi di indirizzamento utilizzabili per indicare l'operando sorgente sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto dell'istruzione MOVE in CCR è:

L'operando sorgente è una word, ma soltanto il byte di ordine basso viene messo nel registro di stato.

Evidentemente, questa istruzione modifica tutti i flag di stato (X, N, Z, V e C).

MOVE in SR (Trasferimento nel Registro di Stato)

Questa istruzione è un caso speciale dell'istruzione MOVE usata per mettere il contenuto dell'operando sorgente nel registro di stato. **Si tratta di un'istruzione privilegiata e può essere eseguita soltanto quando il processore è nel modo Supervisore.** I tipi di indirizzamento che si possono impiegare per indicare l'operando sorgente sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto dell'istruzione MOVE in SR è:

Evidentemente, questa istruzione agisce su tutti i bit del registro di stato.

MOVE da SR (Trasferimento dal Registro di Stato)

Questa istruzione sposta il contenuto del registro di stato a 16 bit nel registro o locazione di destinazione. L'operando destinazione può essere specificato con uno dei seguenti modi di indirizzamento:

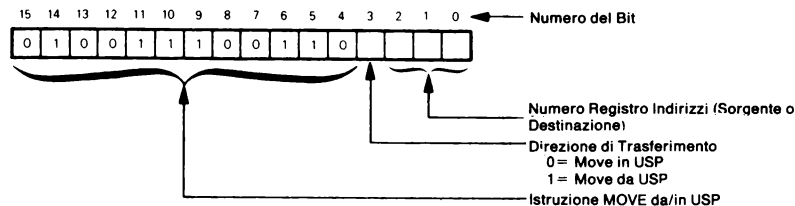
	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione MOVE da SR è:

A differenza dell'istruzione MOVE in SR, questa istruzione può essere eseguita sia in modo Supervisore che in modo Utente. Non modifica il contenuto del registro di stato; si limita a salvarlo nella locazione indicata.

MOVE USP (Trasferimento del Puntatore allo Stack Utente)

Questa istruzione trasferisce il contenuto del puntatore allo stack Utente (A7) dal o nel registro indirizzi specificato. **Si tratta di un'istruzione privilegiata e può essere eseguita solo quando il processore si trova nel modo Supervisore.**

Il codice oggetto dell'istruzione MOVE USP è:

MOVEA (Trasferimento in un Registro Indirizzi)

È una versione speciale dell'istruzione MOVE che mette il contenuto di un determinato operando sorgente in un registro indirizzi. L'operando sorgente può essere indicato con uno dei seguenti modi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi	X	X	001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto dell'istruzione MOVEA è:

La sola limitazione di questa versione dell'istruzione MOVE sta nel fatto che possono essere specificati solo operandi della lunghezza di una word o di una long word, in quanto i registri indirizzi non sono in grado di gestire dati da un byte. Nel caso di un operando di una word, prima di metterlo nel registro indirizzi, ne viene esteso il segno.

L'istruzione MOVEA non agisce su nessuno dei flag del registro di stato.

MOVEM (Trasferimento dalla Memoria ai Registri)

Questa istruzione pone nei registri indicati il contenuto di locazioni di memoria consecutive, a partire dalla locazione corrispondente all'indirizzo effettivo che può essere specificato mediante uno dei seguenti modi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento				
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato				

Il codice oggetto dell'istruzione MOVEM dalla memoria:

Questa istruzione consiste sempre di almeno due word: la prima contiene il codice oggetto dell'istruzione MOVEM e l'indicazione del registro effettivo, mentre la seconda è la maschera dell'elenco dei registri, che indica quelli che dovranno essere utilizzati. Quando uno

dei bit di questa word è posto a uno, significa che il contenuto del registro corrispondente deve essere caricato dalla memoria. **I registri sono caricati a partire dall'indirizzo sorgente specificato, proseguendo verso la parte alta della memoria. L'ordine di trasferimento va dal registro dati D0 (o dal registro dati di numero più basso fra quelli indicati) al registro dati D7, per proseguire fino al registro indirizzi A7.** In altre parole, i registri sono caricati nell'ordine in cui essi appaiono nella maschera, iniziando dal bit 0 e proseguendo fino al bit 15.

La Figura 22-34 mostra l'esecuzione dell'istruzione MOVEM dalla memoria con l'impiego dell'indirizzamento indiretto a registro indirizzi con postincremento. In questa figura, il contenuto dei registri D1, D3, A1, A2 e A3 viene prelevato da cinque word consecutive della memoria dati, a partire dall'indirizzo ppqqqq. Quando con questa istruzione viene indicata una grandezza del dato pari ad una word, ciascuna word subisce l'estensione del segno e nel registro indicato è caricata la long word risultante . Perciò, usando gli operandi della Figura 22-34, una volta eseguita l'istruzione, il registro D1 conterrà 00000101_{16}, mentre il registro A2 conterrà $FFFFF000_{16}$. Se viene indicata una grandezza del dato pari ad una long word, allora verranno usati quattro byte di memoria per caricare ciascuno dei registri indicati.

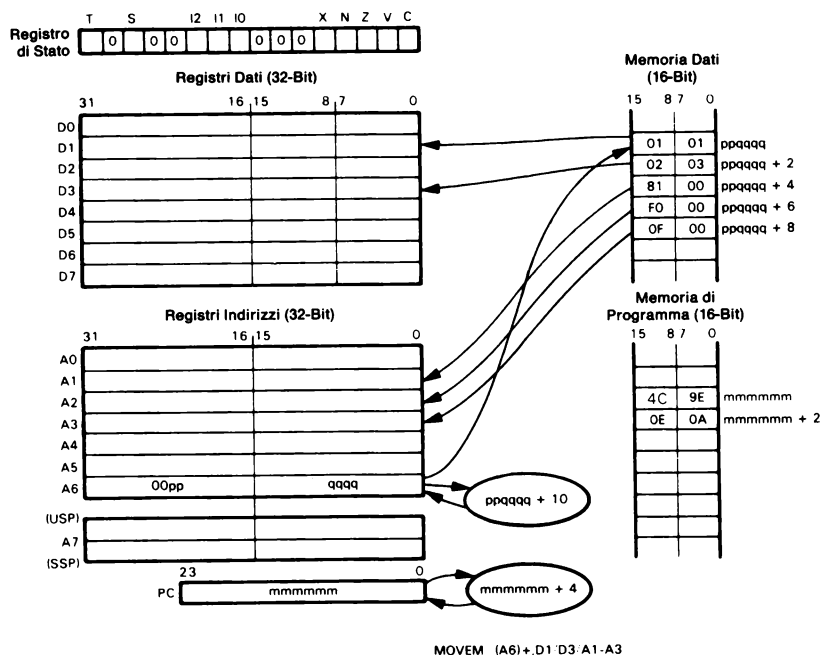



Figura 22-34. Esecuzione dell'Istruzione MOVEM (da Memoria) con Indirizzamento a Registro Indirizzo.

I registri da utilizzare vanno indicati all'assemblatore mediante una notazione particolare. Viene usata una barra (/) per separare i nomi dei vari registri. Il segno meno (-) serve a specificare un gruppo di registri. Perciò, D1/D3 indica i registri D1 e D3, mentre A1-A3 indica i registri A1, A2 e A3.

Usando l'indirizzamento indiretto a registro con postincremento, come nella Figura 22-34, il registro indirizzi incrementato è aggiornato in modo che, una volta che l'istruzione è completata, contenga l'indirizzo dell'ultima word più 2.

L'esecuzione dell'istruzione MOVEM non modifica nessuno dei flag di stato. Questa istruzione si rivela utile per ripristinare in modo rapido ed efficiente il contesto del processore e, perciò, sarà molto utile nella compilazione dei linguaggi ad alto livello.

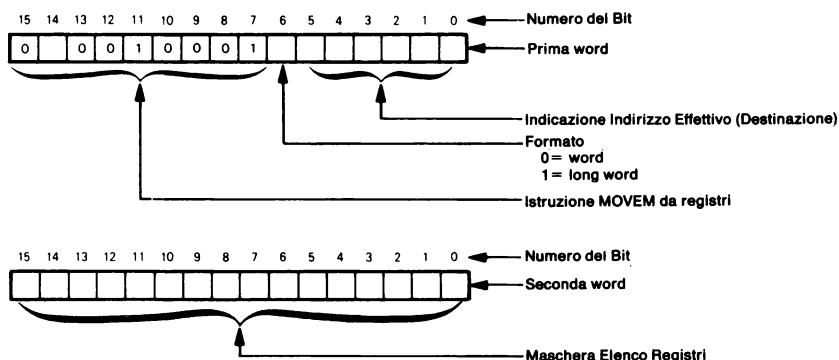
MOVEM (Trasferimento dai Registri alla Memoria)

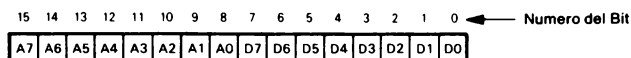
Questa istruzione salva il contenuto dei registri specificati in una serie di locazioni di memoria consecutive. Si tratta, quindi, di un'istruzione complementare rispetto alla precedente.

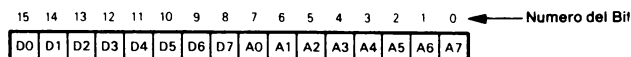
I tipi di indirizzamento che è possibile utilizzare con l'istruzione MOVEM sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento				
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento		X	111	010
Relativo al Contatore di Programma con Indice		X	111	011
Immediato				

La sola differenza rispetto ai tipi di indirizzamento consentiti con la versione "da memoria" sta nel fatto che, in questo caso, si può utilizzare l'indirizzamento indiretto a registro con predecremento (ma non postincremento), mentre nel caso precedente accadeva esattamente l'opposto.

Il codice oggetto per questa versione dell'istruzione MOVEM può essere rappresentato nel modo seguente:

La seconda word dell'istruzione fornisce la maschera per l'elenco dei registri. Se viene utilizzato uno qualsiasi dei vari modi di indirizzamento, tranne quello con predecremento, la maschera è analoga a quella descritta per la versione precedente:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0 ◄— Numero del Bit
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Tuttavia, se viene usato l'indirizzamento con predecremento la maschera dei registri sarà:

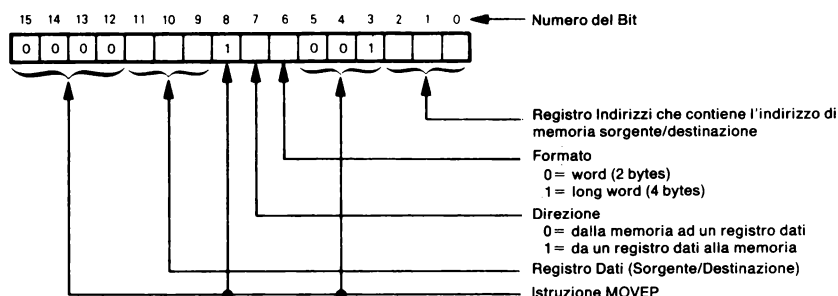
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0 ◄— Numero del Bit
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

In questo caso, i registri sono salvati a partire dall'indirizzo indicato -2 e procedendo verso la parte bassa della memoria. Si inizia con il registro A7 per arrivare al registro A0 e proseguendo con il registro D7 fino al D0. Questo era scontato, in quanto gli altri tipi di indirizzamento usano indirizzi via via crescenti, mentre nel caso del predecremento avviene l'esatto contrario. L'assemblatore, comunque, genera le maschere automaticamente.

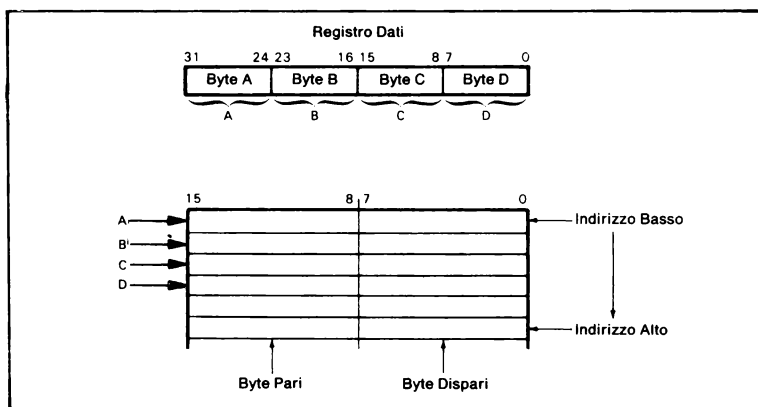
Analogamente a quanto descritto per l'altra versione dell'istruzione MOVEM, saranno salvati solo quei registri, i cui corrispondenti bit della maschera sono posti a 1. Se viene usato l'indirizzamento con predecremento, il registro indirizzi decrementato sarà aggiornato in modo da contenere, al completamento dell'istruzione, l'indirizzo dell'ultima word salvata.

Nessuno dei flag di stato viene modificato dall'istruzione MOVEM.

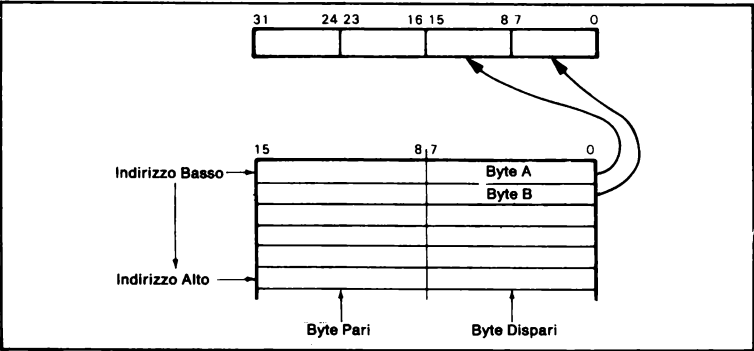
MOVEP (Trasferimento di Dati Periferici)

Questa istruzione è una versione speciale dell'istruzione MOVE e trasferisce da due a quattro byte di dati fra un determinato registro dati e locazioni di memoria alternate. **Ha lo scopo di semplificare il trasferimento di dati fra il processore e dei dispositivi ad 8 bit. L'unico tipo di indirizzamento possibile è quello indiretto a registro indirizzi con spostamento.**

Il codice oggetto dell'istruzione MOVEP è:

Come potete constatare, la direzione del trasferimento dei dati può essere dalla memoria ad un registro dati oppure da un registro dati alla memoria. Il codice oggetto dell'istruzione è seguito da un valore di spostamento a 16 bit, che sarà sommato al contenuto di un determinato registro indirizzi, per ottenere l'indirizzo dell'operando, sorgente o destinazione, in memoria.

Se l'ordine di grandezza dell'operazione è di una long word, allora sono trasferiti quattro byte di 8 bit ciascuno. Ad esempio, se l'istruzione MOVEP è usata per trasferire un dato da un registro dati alla memoria ed è specificata una grandezza di una long word, il trasferimento avviene nel modo seguente:

Il byte di ordine alto del registro dati è trasferito per primo, quello di ordine basso per ultimo. **Ogni volta che viene trasferito un byte, l'indirizzo viene incrementato di 2. (Si noti che il contenuto del registro indirizzi non viene incrementato).** Perciò, se l'indirizzo di partenza è un numero pari (come nell'illustrazione precedente), tutti i trasferimenti avvengono sulla metà di ordine alto del bus dati. Se invece è un numero dispari sarà utilizzata la metà di ordine basso.

Qualora con l'istruzione MOVEP sia specificato un formato di una

word, viene trasferita la metà di ordine basso del registro, a cominciare dal byte più alto:

Questa illustrazione mostra il trasferimento di un dato dalla memoria ad un registro dati. Il primo byte viene caricato nei bit 8-15 del registro dati, il secondo nei bit 0-7. In questo caso, è stato specificato un indirizzo dispari.

La figura 22-35 mostra l'esecuzione dell'istruzione MOVEP, con i dati che sono trasferiti da byte pari di memoria al registro dati D4.

Nella figura 22-35, il contenuto del registro A4 è sommato al valore di spostamento a 16 bit (0420_{16}), che segue la word d'istruzione nella memoria di programma. Si presume che l'indirizzo risultante (sssttt) sia un indirizzo pari ed indichi il primo byte dei dati da trasferire. Quel byte ed i tre dei successivi indirizzi pari sono trasferiti nel registro D4.

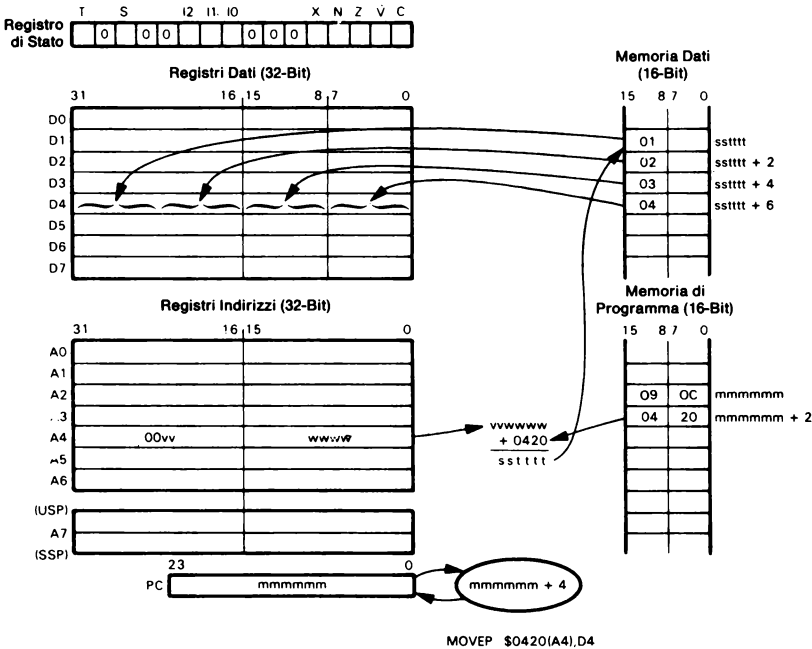
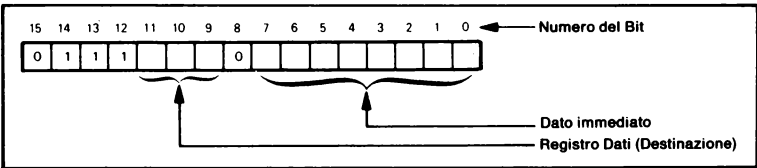



Figura 22-35. Esecuzione dell'Istruzione MOVEP.

L'istruzione MOVEP non agisce su nessuno dei flag di stato.

MOVEQ (Trasferimento di un Dato Immediato)

Questa istruzione sposta il dato immediato contenuto nel codice oggetto dell'istruzione in un determinato registro dati. Il codice oggetto è il seguente:

Gli otto bit meno significativi della word d'istruzione contengono il dato immediato, che, mediante estensione del segno, viene trasformato in un operando lungo e tutti i 32 bit sono trasferiti nel registro dati.

La figura 22-36 mostra l'esecuzione dell'istruzione MOVEQ. In questo caso, il dato immediato ($7F_{16}$) subisce l'estensione del segno ed è caricato nel registro D3. Perciò, dopo l'esecuzione dell'istruzione, il registro D3 conterrà $0000007F_{16}$.

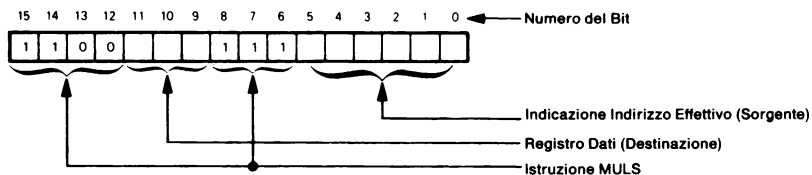
Il flag di Negativo (N) diventa uno se il valore caricato nel registro dati è negativo, altrimenti sarà azzerato. Il flag di Zero (Z) verrà posto a uno se sul registro dati viene caricato uno zero; in caso contrario viene anch'esso azzerato. I flag di Overflow (V) e di Carry (C) sono sempre azzerati. Il flag di Extend (X) resta invariato.

MULS (Moltiplicazione con Segno)

Questa istruzione moltiplica due operandi a 16 bit, con segno, fornendo un risultato a 32 bit anch'esso con segno. La moltiplicazione viene eseguita usando l'aritmetica binaria in complemento a due. L'operando destinazione deve sempre essere un registro dati, mentre l'operando sorgente può essere indicato con uno dei seguenti tipi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Figura 22-36. Esecuzione dell'Istruzione MOVEQ.

Il codice oggetto dell'istruzione MULS è:

La Figura 22-37 mostra l'esecuzione dell'istruzione MULS, usando l'indirizzamento indiretto a registro indirizzi, per specificare l'operando sorgente. Dopo che è stata eseguita questa istruzione, il registro D2 conterrà il prodotto (00800000_{16}).

Il flag di Negativo (N) sarà posto a uno se il risultato è negativo; verrà azzerato in caso contrario. Il flag di Zero (Z) diventa uno quando il risultato è zero, altrimenti viene azzerato. I flag di Carry (C) e di Overflow (V) sono sempre azzerati. Il flag di Extend (X) resta immutato.

Si noti che il valore che viene prelevato dal registro dati corrisponde alla word di ordine basso del registro stesso. La word di ordine alto del registro destinazione non è utilizzata nella moltiplicazione ed il valore in essa contenuto va perso, quando nel registro viene messo il prodotto risultante, che è un valore a 32 bit.

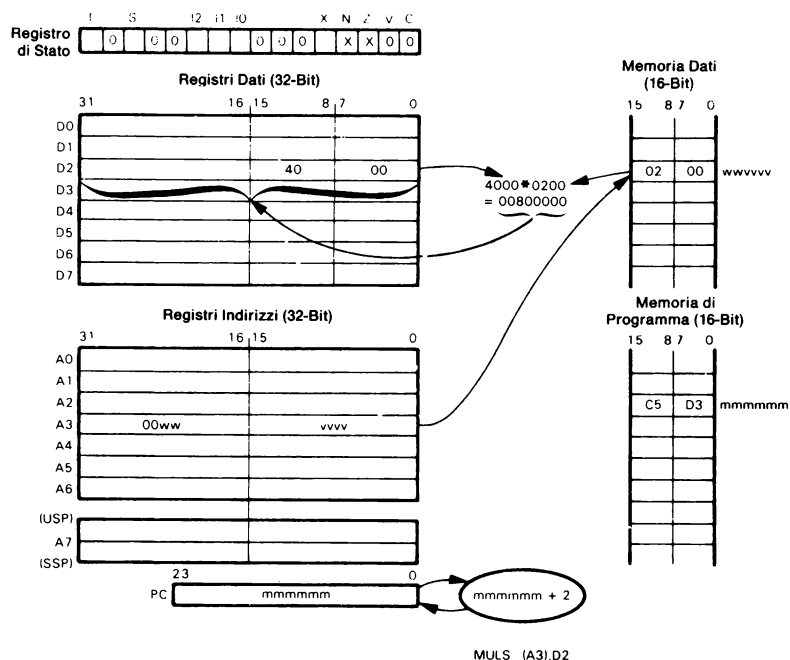



Figura 22-37. Esecuzione dell'Istruzione MULS con Indirizzamento Indiretto a Registro Indirizzi.

MULU (Moltiplicazione senza Segno)

Questa istruzione moltiplica due operandi a 16 bit sprovvisti di segno fornendo un risultato a 32 bit. L'operazione è eseguita utilizzando l'aritmetica binaria senza segno. L'operando destinazione deve essere sempre un registro dati, che servirà anche a contenere il prodotto risultante. Per indicare l'operando sorgente è possibile utilizzare uno dei seguenti tipi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto per l'istruzione MULU è:

L'operando a 16 bit prelevato dal registro dati corrisponde alla word di ordine basso del registro stesso; il valore contenuto nella word di ordine alto non viene utilizzato ai fini della moltiplicazione e va perso, quando nel registro vieno messo il prodotto risultante a 32 bit.

L'esecuzione dell'istruzione MULU è analoga a quella dell'istruzione MULS, tranne, naturalmente, per il fatto che, in questo caso, è utilizzata l'aritmetica senza segno. Consultate la Figura 22-37 per maggiori dettagli sull'esecuzione dell'istruzione MULS.

Il flag di Negativo (N) diventa uno, se il bit più significativo del prodotto è uno, altrimenti viene azzerato. Il flag di Zero (Z) èposto a uno, se il prdotto è zero ed è azzerato, in caso contrario. I flag di Carry (C) e di Overflow (V) sono sempre azzerati. Il flag di Extend (X) resta invariato.

NBCD (Negazione Decimale con Extend)

Questa istruzione sottrae l'operando destinazione ed il valore del flag di Extend (X) da zero. il risultato viene salvato nella locazione di destinazione. L'operazione è eseguita in aritmetica decimale codificata binaria (BCD). I modi di indirizzamento possibili sono i seguenti:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione NBCD è:

La Figura 22-38 mostra l'esecuzione dell'istruzione NBCD, con l'impiego dell'indirizzamento indiretto a registro indirizzi. L'istruzione NBCD agisce sempre su un dato della grandezza di un byte.

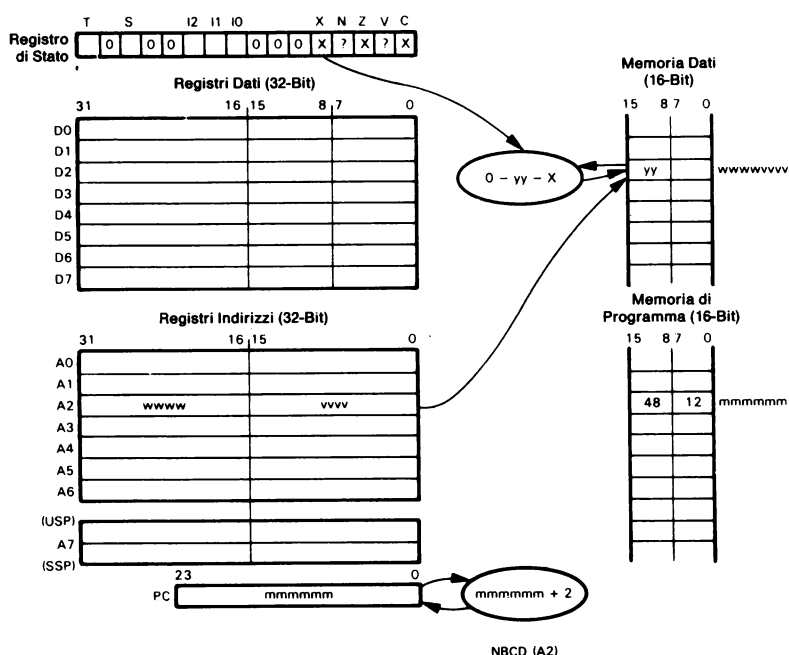



Figura 22-38. Esecuzione dell'Istruzione NBCD con Indirizzamento indiretto a Registro Indirizzi.

Questa istruzione può essere usata in operazioni multibyte, per calcolare il negativo di un dato BCD. Il Capitolo 8 fornisce una descrizione della aritmetica BCD multibyte. Questa istruzione agisce sui flag di stato in maniera analoga all'istruzione SBCD.

NEG (Negazione)

Questa istruzione sottrae il contenuto dell'operando destinazione da zero usando l'aritmetica binaria in complemento a due. Il risultato viene messo nellà locazione (o registro) di destinazione.

L'operando può essere indicato mediante uno dei seguenti tipi di indirizzamento:

Modi di Indirizzamento Possibili	Operando Sorgente	Operando Destinazione	Campo Ind.Eff. Destinazione Modo	Campo Ind.Eff. Destinazione Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto ell'istruzione NEG è:

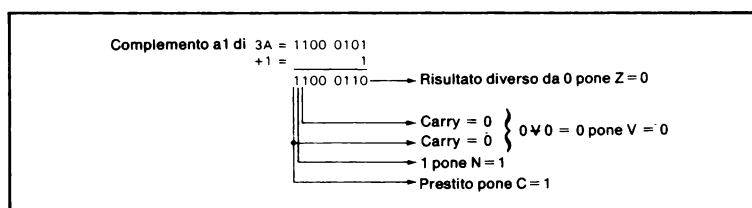
La figura 22-39 mostra l'esecuzione dell'istruzione NEG, con l'indirizzamento diretto a registro dati. Se, ad esempio, il byte meno significativo del registro D3 contiene inizialmente $3A_{16}$, dopo che il processore ha eseguito l'istruzione NEG.B D3 conterrà $C6_{16}$.

I flag di stato (N, Z, V, C, X) sono modificati in modo analogo a quanto accade con l'istruzione SUB.

Figura 22-39. Esecuzione dell'Istruzione NEG con Indirizzamento Diretto a Registro Indirizzi.

NEGX (Negazione con Extend)

Questa istruzione sottrae il contenuto dell'operando destinazione ed il valore del flag di Extend (X) da zero. Il risultato viene messo nella locazione (o registro) di destinazione. I modi di indirizzamento che è possibile utilizzare per indicare l'operando destinazione sono:

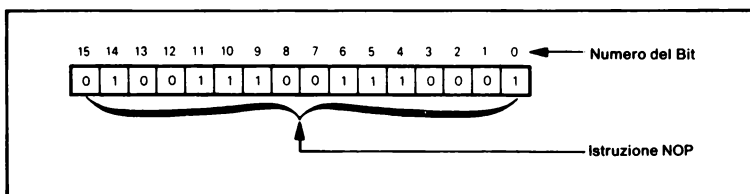
	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione NEGX è:

Questa istruzione agisce in modo del tutto analogo all'istruzione NEG, tranne per il fatto che, in questo caso, da zero viene sottratto anche il contenuto del flag X. Perciò, essa si rivela molto utile per eseguire le operazioni in precisione multipla, ampiamente descritte nel Capitolo 8.

I flag di stato sono posti a uno o azzerati in base agli stessi criteri indicati per l'istruzione SUBX.

NOP (Nessuna Operazione)

NOP è un'istruzione che si limita ad incrementare il contatore di programma. Il relativo codice oggetto è:

L'esecuzione dell'istruzione NOP è illustrata nella Figura 22-40.

L'istruzione NOP viene usata soprattutto nei seguenti casi:

1. Per assegnare un valore ad una label senza interferire con il programma oggetto.
2. Per realizzare un determinato intervallo di tempo.
3. Per sostituire istruzioni, che, in seguito a correzioni e modifiche, non sono più necessarie.
4. In fase di debugging, per sostituire istruzioni (come la chiamata di una subroutine), che non ci interessa esaminare.

L'istruzione NOP è usata raramente in programmi definitivi, ma viene impiegata soprattutto in fase di debugging e di collaudo.

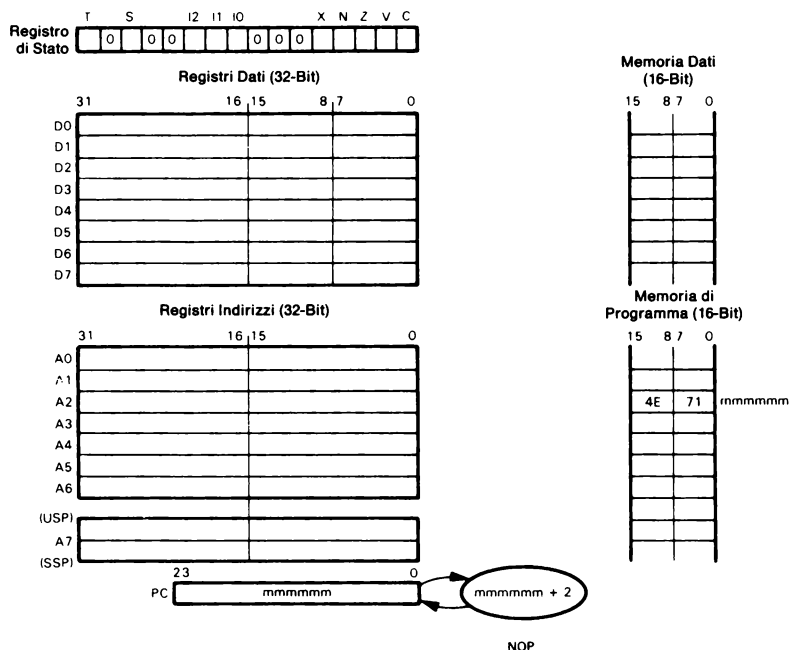



Figura 22-40. Esecuzione dell'Istruzione NOP.

NOT (Complemento Logico)

Questa istruzione esegue un complemento, bit per bit, del contenuto dell'operando destinazione. Il risultato viene memorizzato nella locazione (o registro) di destinazione. Si tratta di un'operazione di complemento a uno, che sostiuisce gli 1 dell'operando con degli 0 e viceversa.

I tipi di indirizzamento utilizzabili per indicare l'operando destinazione sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione NOT è:

La Figura 22-41 mostra l'esecuzione dell'istruzione NOT, con l'indirizzamento indiretto a registro indirizzi.

Il flag di Negativo (N) è posto a uno se il risultato è negativo ed è azzerato in caso contrario. Il flag di Zero (Z) diventa uno se il risultato è zero, altrimenti viene azzerato. I flag di Overfoilw (V) e di Carry (C) sono sempre azzerati. Il flag di Extend (X) resta immutato.

OR (OR Logico Inclusivo)

Questa istruzione esegue un OR logico (inclusivo), bit per bit, del contenuto dell'operando sorgente con quello dell'operando destinazione e salva il risultato nella locazione di destinazione.

Esistono due forme fondamentali per questa istruzione. Nella prima, è un registro dati che costituisce uno degli operandi e anche la locazione dove sarà messo il risultato. In questo caso, sono consentiti tutti i modi di indirizzamento, tranne quello diretto a registro indirizzi:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Nell'altra forma dell'istruzione OR, un registro dati rappresenta l'operando sorgente, mentre l'operando destinazione può essere indicato mediante uno dei seguenti tipi di indirizzamento:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X	X	000	rrr
Diretto a Registro Indirizzi			00	rrr
Indiretto a Registro Indirizzi		X	01	rrr
Indiretto a Registro con Postincremento		X	01	rrr
Indiretto a Registro con Predecremento		X	10	rrr
Indiretto a Registro con Spostamento		X	10	rrr
Indiretto a Registro con Indice		X	11	rrr
Assoluto Corto		X	11	000
Assoluto Lungo		X	11	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

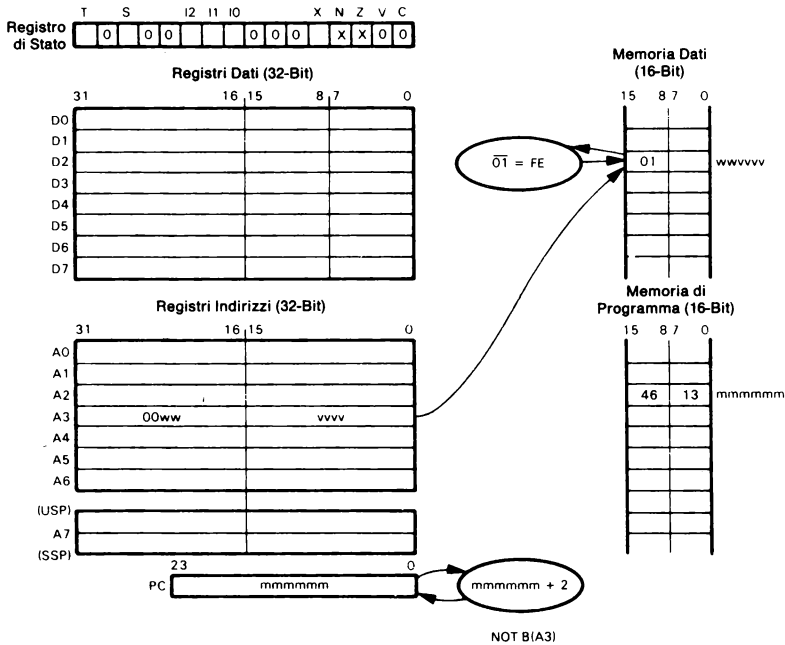
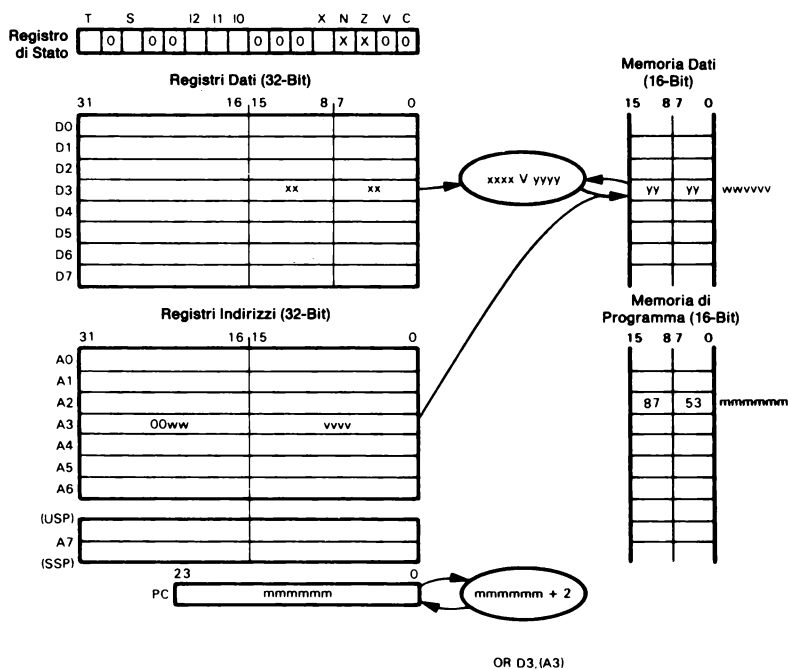
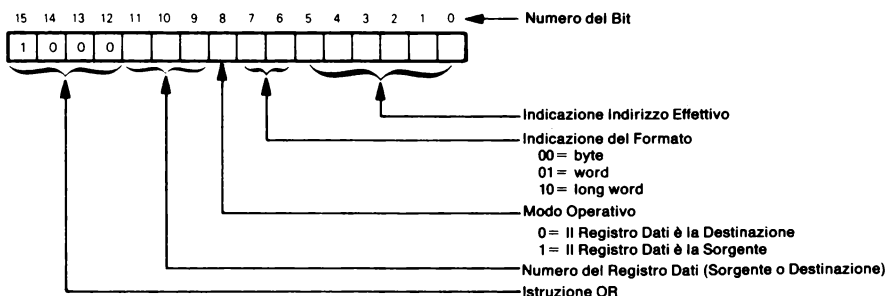



Figura 22-41. Esecuzione dell'Istruzione NOT con Indirizzamento Indiretto a Registro Indirizzi.

Figura 22-42. Esecuzione dell'Istruzione OR con Indirizzamento Indiretto a Registro Indirizzi.

Il codice oggetto dell'istruzione OR è:

La figura 22-42 mostra l'esecuzione dell'istruzione OR, con indirizzamento indiretto a registro indirizzi. In questa figura, il registro indirizzi A3 indica la locazione dell'operando (wwvvvv) e la locazione sorgente è rappresentatta dai 16 meno significativi del registro D3.

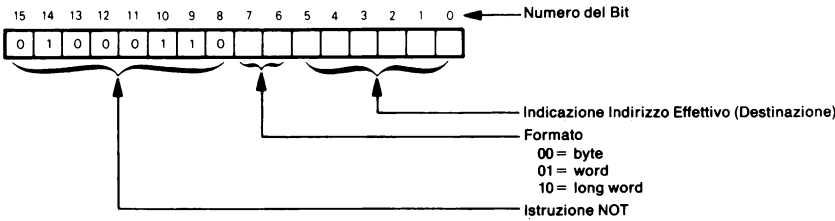
OR è un'istruzione logica molto comune, usata spesso per porre a 1 determinati bit. Eseguendo, infatti, l'OR logico di un bit con 1 si otterrà un risultato di 1, mentre l'OR di un bit con 0 lascia il bit al suo valore iniziale.

ORI (OR Inclusivo Immediato)

Questa istruzione viene usata per eseguire l'OR di un dato immediato, presente nelle successive locazioni della memoria di programma, con l'operando destinazione, al posto del quale viene messo il risultato. I tipi di indirizzamento utilizzabili per indicare l'operando destinazione sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	X			
Registro di Stato		X	111	100

Si noti che l'operando destinazione può essere rappresentato dai codici di condizione o dall'intero registro di stato. **In quest'ultimo caso, si tratterà di un'istruzione privilegiata e potrà essere eseguita solamente quando il processore si trova nel modo Supervisore.**

Il codice oggetto dell'istruzione ORI è:

Il formato dell'istruzione ORI può essere di un byte, di una word o di una long word, a seconda di quanto specificato. Il dato immediato segue la word d'istruzione e deve corrispondere al formato indicato. Perciò, una o due word di dato immediato seguiranno il codice operativo dell'istruzione nella memoria di programma. Se l'istruzione indica un operando di un byte, verrà utilizzato il byte di ordine basso (secondo) della word di dato immediato. L'assemblatore seleziona, automaticamente, il byte corretto. Se l'istruzione si riferisce al registro di stato e la grandezza dell'operazione è di un byte, l'operando destinazione è il byte di ordine basso del registro di stato, quello contenente i codici di condizione. Se la grandezza

dell'operazione è di una word, l'operando destinazione è l'intero registro di stato e si tratterà di un'istruzione privilegiata.

L'esecuzione dell'istruzione ORI è la stessa illustrata per l'istruzione OR nella Figura 22-42, tranne per il fatto che il dato immediato è contenuto nella memoria di programma subito dopo la word dell'istruzione.

I flag N e Z sono modificati dall'istruzione ORI, in base al risultato ottenuto. I flag di Overflow (V) e di Carry (C) saranno sempre azzerati. Il flag di Extend (X) resta invariato. Naturalmente, se la destinazione dell'istruzione è costituita dai codici di condizione o dall'intero registro di stato, allora sarà l'esecuzione stessa dell'operazione a causarne la modifica.

PEA (Caricamento di un Indirizzo Effettivo nello Stack)

Questa istruzione forma un indirizzo effettivo, usando uno dei tipi di indirizzamento di controllo, e, quindi, lo pone sullo stack. Gli indirizzamenti possibili sono:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento				
Indiretto a Registro con Predecremento				
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato				

Il codice oggetto dell'istruzione PEA è:

La Figura 22-43 illustra l'esecuzione dell'istruzione PEA, con l'indirizzamento assoluto corto. L'indirizzo corto a 32 bit (420_{16}) è trasformato in un valore a 16 bit, mediante l'estensione del segno. Questo indirizzo a 32 bit viene posto allo stack di sistema, utilizzando, in questo caso, il puntatore allo stack Utente (USP).I bit meno significativi dell'indirizzo effettivo (0420_{16}) sono messi nella word di memoria wwwwvvvv - 2. Quindi, il puntatore allo stack è decrementato di 2 ed i 16 bit più significativi dell'indirizzo sono depositati

sullo stack. Il valore finale del puntatore allo stack sarà inferiore di quattro rispetto al valore iniziale.

Naturalmente, se vengono utilizzati dei tipi di indirizzamento diversi saranno necessari calcoli più complessi per ottenere l'indirizzo.

L'indirizzo che viene messo sullo stack dall'istruzione PEA è sempre un valore a 32 bit, indipendentemente dai tipi di indirizzamento utilizzati.

Nessun flag di stato è modificato dall'istruzione PEA.

RESET (Reset di Dispositivi Periferici)

Questa istruzione genera un impulso in uscita sul pin di RESET del processore ed è, quindi, utilizzata per eseguire il Reset di dispositivi periferici. Lo stato del processore non subisce modifiche, se si eccettua l'incremento di 2 del contatore di programma. L'esecuzione del programma continua, perciò, con l'istruzioni immediatamente successiva. Il codice oggetto dell'istruzione RESET è il seguente:

Figura 22-43. Esecuzione dell'Istruzione PEA con Indirizzamento Assoluto Corto.

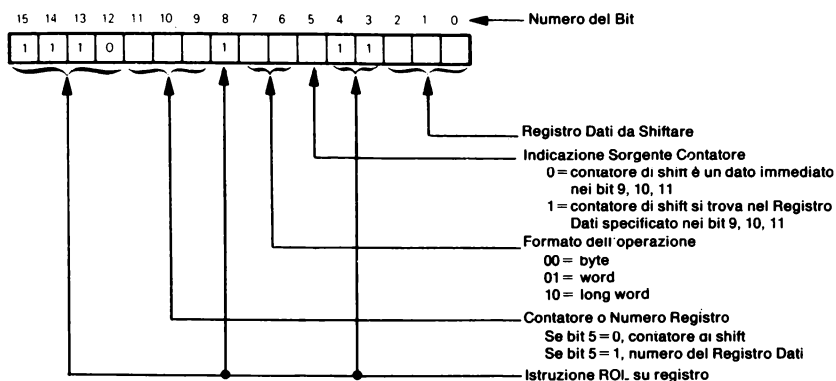
Nessuno dei flag di stato è influenzato dall'esecuzione dell'istruzione RESET.

ROL (Rotazione a Sinistra in un Registro Dati)

Questa istruzione provoca la rotazione a sinistra del contenuto di un determinato registro dati. Il flag di Carry (C) riceve l'ultimo bit uscito dalla estremità più significativa del registro dati, che va ad occupare anche la posizione di ordine basso del registro stesso. Il numero di shift può essere specificato in un altro registro dati o mediante un dato immediato.

Se il numero di rotazioni è contenuto in un registro dati, esso occupa i sei bit meno significativi e si tratterà dunque di un valore compreso fra 0 e 63 (modulo 64).

Se viene usato un dato immediato, può essere specificato un numero di shift compreso fra 1 e 8, all'interno del codice operativo dell'istruzione.

Il codice oggetto dell'istruzione ROL è:

Il bit 5 del codice operativo dell'istruzione stabilisce se il numero di shift è indicato tramite un dato immediato contenuto nei bit 9, 10 e 11 oppure si trova nel registro dati, indicato sempre dai bit 9, 10 e 11.

La Figura 22-44 mostra l'esecuzione di un istruzione ROL, con il registro D0 come operando destinazione e con il numero di shift nel registro D3.

Se il registro D0 contiene $AD1F_{16}$ ed il registro D3 contiene 03_{16}, il valore presente nel registro D0 viene ruotato di tre posizioni verso sinistra. Dopo l'esecuzione dell'istruzione ROL D3,D0, il registro D0 contiene $68FD_{16}$ ed il flag di Carry (C) vale 1.

I bit usciti dall'estremità di ordine alto vanno nel flag di Carry e nel bit di ordine basso del registro, come meglio chiarisce lo schema seguente:

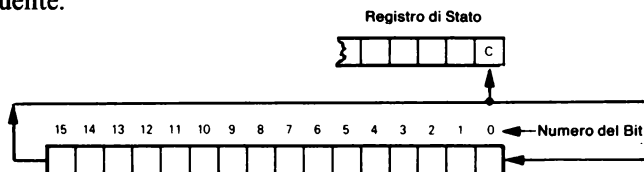
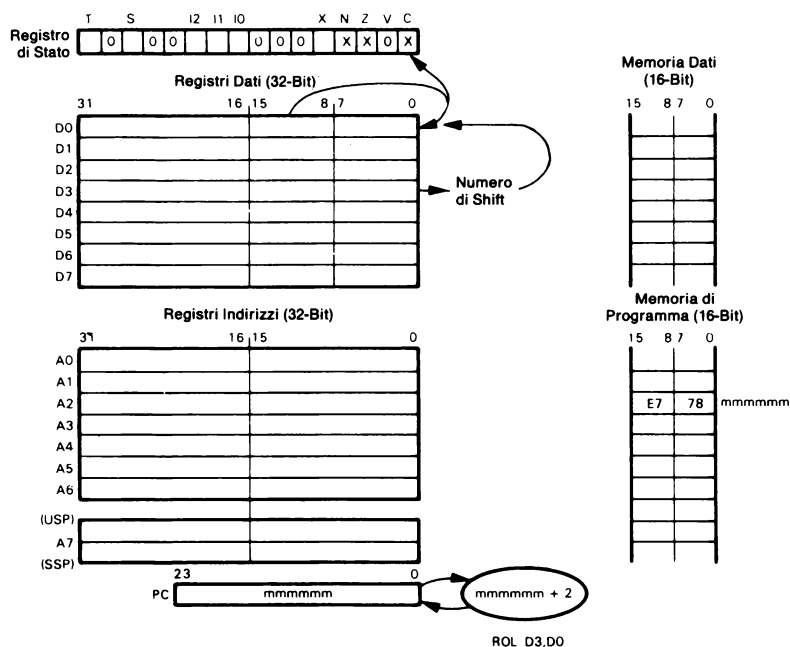



Figura 22-44. Esecuzione dell'Istruzione ROL con Operando e Numero di Shift nel Registro Dati.

Il flag di Negativo (N) diventa uno se il bit più significativo del risultato è 1, altrimenti viene azzerato. Il flag di Zero (Z) è posto a uno, se il risultato è zero ed è azzerato in caso contrario. Il flag di Overflow (V) è sempre azzerato. Il flag di Carry (C) assumerà il valore dell'ultimo bit uscito dall'operando, in seguito allo shift; verrà azzerato, se il numero di shift è 0. Il flag di Extend (X) resta invariato.

La Figura 22-44 mostra un'operazione ROL, con un operando della grandezza di una word all'interno del registro dati. Sono interessati soltanto i bit da 0 a 15; gli altri (16-31) restano invariati. L'istruzione ROL può agire anche su operandi di un byte o di una long word, contenuti sempre in un registro dati.

A proposito dell'istruzione descritta nella Figura 22-44, gli stessi risultati si potevano ottenere anche nel modo seguente:

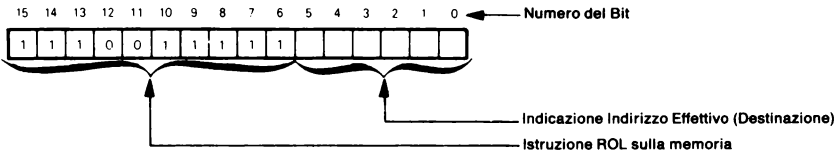
```
ROL #3,D0
```

In questo caso viene impiegato un dato immediato per indicare il numero di shift, invece di utilizzare il contenuto di un altro registro dati come in precedenza. Il vantaggio derivante dall'impiego di un registro dati consiste nella possibilità di modificarne il contenuto durante l'esecuzione del programma, mentre un dato immediato si trova nella memoria di programma e perciò, di solito, non è modificabile.

ROL (Rotazione a Sinistra di una Word di Memoria)

Esegue la stessa operazione dell'istruzione precedente, ma agisce su un operando contenuto in memoria, invece che in un registro dati. Questa versione dell'istruzione ROL presenta due limitazioni: la grandezza dell'operando non può essere superiore ad una word e lo shift può essere di una sola posizione. I tipi di indirizzamento possibili con questa versione dell'istruzione ROL sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

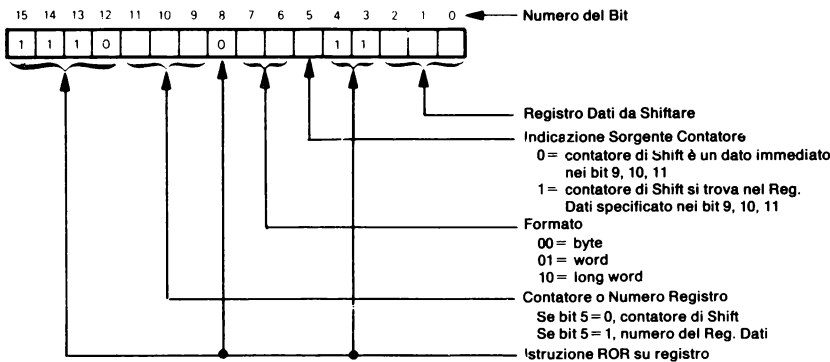
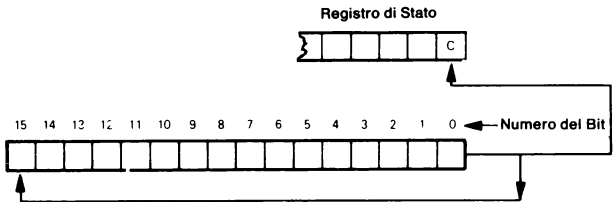
Il codice oggetto di questa istruzione è il seguente:

L'esecuzione di questa istruzione ROL è analoga a quella illustrata per la versione a registro dati. L'operando in memoria è ruotato a sinistra di un bit ed il bit uscito dall'estremità di ordine alto va sia nel flag di Carry (C) del registro di stato che nel bit zero della word di memoria.

Vi rimandiamo all'istruzione ROL a registro per una descrizione degli effetti di questa istruzione sui flag di stato.

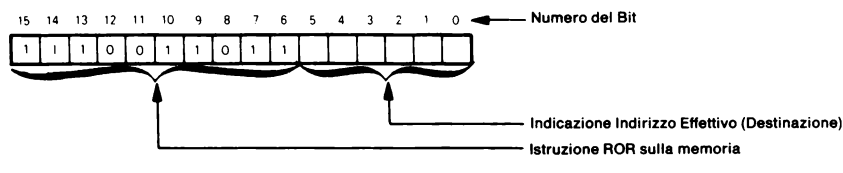
ROR (Rotazione a Destra)

Questa istruzione ruota verso destra i bit di un determinato operando. Come per le istruzioni ROL descritte nelle pagine precedenti, esistono due versioni generali di questa istruzione. La prima permette di eseguire lo shift di un operando presente in un registro dati del numero di posizioni indicato dal contenuto di un altro

registro dati o da un dato immediato che fa parte della word d'istruzione. L'altra versione consente di shiftare di una posizione un operando della grandezza di una word contenuto in memoria. I tipi di indirizzamento possibili sono gli stessi elencati per le istruzioni ROL.

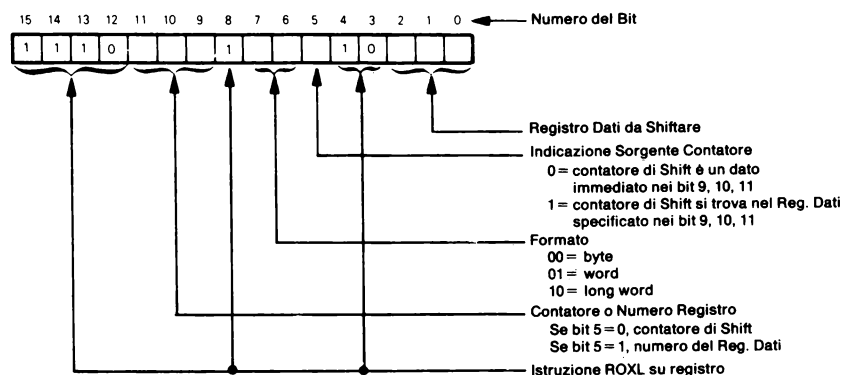
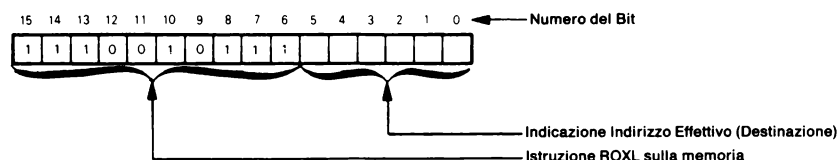
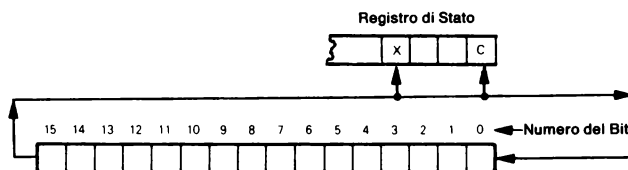
Il codice oggetto dell'istruzione ROR a registro può essere rappresentato nel modo seguente:

Quando viene eseguita l'istruzione ROR, l'operando subisce uno shift verso destra del numero di posizioni indicato ed i bit di ordine basso vanno sia nel flag di Carry (C) del registro di stato che nei bit di ordine alto dell'operando, come appare dallo scheme seguente:

Come nel caso dell'istruzione ROL, la sola differenza tra la versione a registro e quella a memoria sta nel fatto che la seconda deve avere un operando della grandezza di una word e lo shift può essere di una sola posizione. I tipi di indirizzamento consentiti sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati				
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto per la versione sulla memoria dell'istruzione ROR è:

ROXL (Rotazione a Sinistra con Extend)

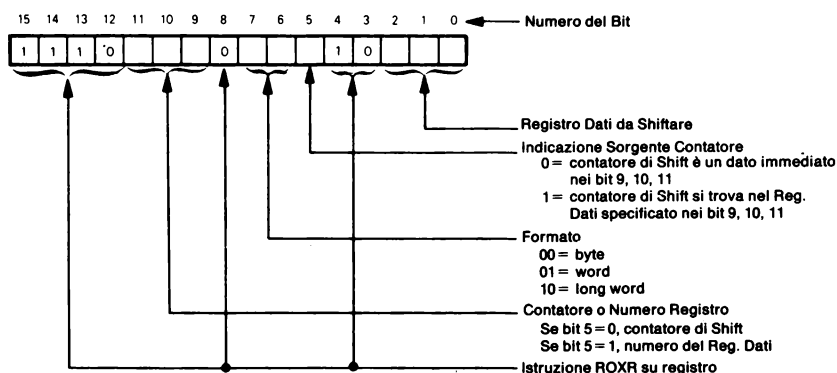
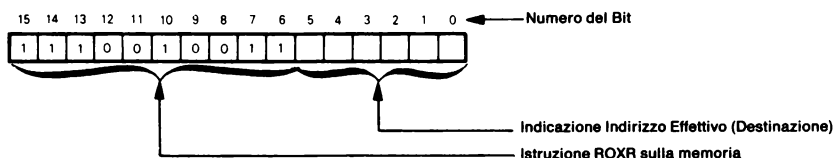
Questa istruzione è analoga all'istruzione ROL, tranne per il fatto che i bit usciti dalla posizione più significativa non vanno soltanto nel flag di Carry (C) e nel bit meno significativo dell'operando, ma anche nel flag di Extend (X). Perciò, **questa versione dell'istruzione di rotazione può essere usata nelle operazioni in precisione multipla.**

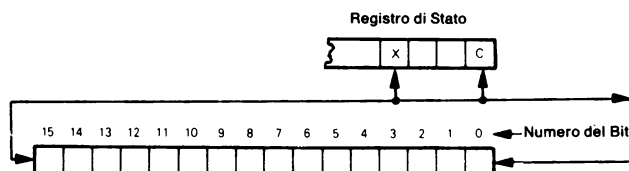
Con questa istruzione sono possibili le versioni ed i modi di indirizziamento indicati per l'istruzione ROL descritta nelle pagine precedenti ed alla quale vi rimandiamo.

Il codice oggetto per la versione su registro dell'istruzione ROXL è:

Il codice oggetto per la versione sulla memoria è invece il seguente:

Per una descrizione dettagliata del funzionamento dell'istruzione ROXL vi rimandiamo a quanto detto a proposito dell'istruzione ROL. Gli effetti di un'istruzione ROXL sono quelli indicati nello schema seguente:

ROXR (Rotazione a Destra con Extend)

Questa istruzione agisce in modo del tutto identico alla istruzione ROR, tranne per il fatto che i bit usciti dalla posizione di ordine basso dell'operando non vanno soltanto nel flag di Carry (C) e nel bit di ordine alto dell'operando, ma anche nel flag di Extend (X) del registro di stato. Perciò, **questa versione della istruzione di rotazione può essere usata nelle operazioni in precisione multipla.**

Con questa istruzione sono possibili le versioni ed i tipi di indirizzamento indicati per l'istruzione ROR, cui vi rimandiamo.

Il codice oggetto dell'istruzione ROXR su registro è:

Il codice oggetto della versione sulla memoria è:

Dal momento che il funzionamento dell'istruzione ROXR è analogo a quello dell'istruzione ROR vi rimandiamo alla decsrizione di

quest'ultima per maggiori dettagli. L'effetto dell'istruzione ROXR su un operando è il seguente:

RTE (Ritorno da una Exception)

Questa istruzione ripristina lo stato di un programma interrotto da un'Exception, caricando il registro di stato ed il contatore di programma dallo stack di sistema. **RTE dovrebbe essere l'ultima istruzione eseguita da una routine destinata a servire un'Exception.**

Il codice oggetto dell'istruzione RTE è:

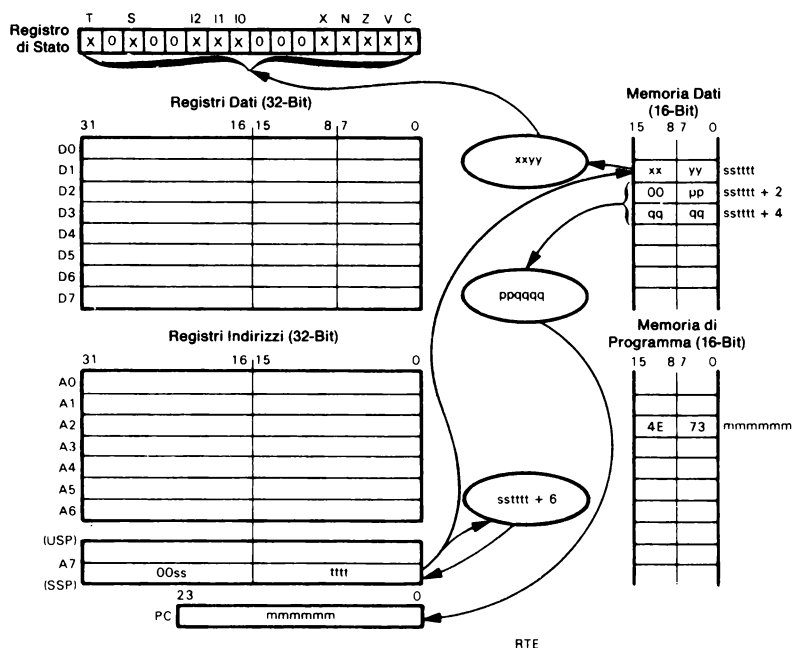
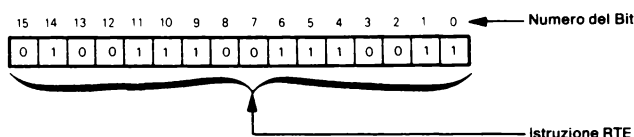



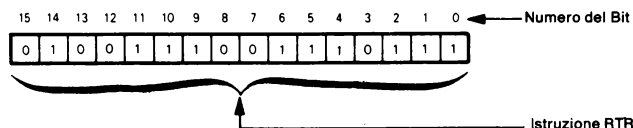
Figura 22-45. Esecuzione dell'Istruzione RTE.

La Figura 22-45 illustra l'esecuzione dell'istruzione RTE. Per prima cosa, la word prelevata dalla sommità dello stack viene caricata nel registro di stato, sostituendo il precedente valore. Le due word successive prelevate dallo stack sono caricate nel contatore di programma e l'esecuzione continuerà, appunto, a partire da questo nuovo indirizzo.

Il puntatore allo stack utilizzato durante l'istruzione RTE è quello allo stack Supervisore (SSP), dal momento che l'elaborazione di un'Exception avviene sempre in modo Supervisore. Una volta eseguita l'istruzione RTE, il processore può trovarsi nel modo Supervisore o Utente, a seconda dello stato del bit S del registro di stato. Evidentemente, tutti i 16 bit di questo registro, cioè il byte di Sistema e quello Utente, sono interessati dalle RTE, fatta eccezione per quei bit non ancora usati che rimangono sempre a 0.

RTR (Ritorno e Ripristino dei Codici Condizione)

Questa istruzione ripristina la parte del registro di stato relativa ai codici di condizione, prelevando una word dallo stack e mettendone i cinque bit meno significativi nel registro di stato. Restituisce, quindi, il controllo da un subroutine al programma chiamante, prelevando l'indirizzo di ritorno dallo stack e mettendolo nel contatore di programma.

Il codice oggetto dell'istruzione RTR è:

La Figura 22-46 mostra l'esecuzione dell'istruzione RTR. I cinque bit meno significativi della prima word prelevata dallo stack di sistema sono usati per rimpiazzare il precedente contenuto dei codici di condizione nel registro di stato (i flag X, N, Z,V e C). Le due successive word prelevate dallo stack di sistema sono caricate nel contatore di programma, il cui precedente contenuto va perduto. Dopo aver prelevato ciascuna word dallo stack, il processore incrementa il puntatore, per cui il valore finale di questo sarà maggiore di sei rispetto al valore iniziale. Nella Figura 22-45 era utilizzato il puntatore allo stack Utente (USP). Se il processore si fosse trovato nel modo Supervisore, sarebbe stato usato, invece, il puntatore allo stack Supervisore (SSP).

L'istruzione RTR modifica soltanto i cinque bit meno significativi del registro di stato; il byte di sistema resta inalterato. Questa è la sola differenza tra l'istruzione RTR e l'istruzione RTE, che sostituisce l'intero contenuto dei 16 bit del registro di stato.

L'MC68000 non dispone di una speciale istruzione di salto ad una subroutine, che salvi automaticamente i codici di condizione sullo stack. Perciò, se avete intenzione di usare l'istruzione RTR, è la vostra subroutine che deve provvedere a salvare i codici di condizione. A questo scopo può essere utilizzata l'istruzione MOVE da SR con indirizzamento indiretto a registro, indicando A7 come registro:

```
MOVE SR,-(A7)
```

Questa istruzione deve essere eseguita prima che la subroutine utilizzi lo stack di sistema per altri scopi, in quanto l'istruzione RTR si aspetta che il valore dei codici di condizione e quello del contatore di programma occupino tre word consecutive dello stack.

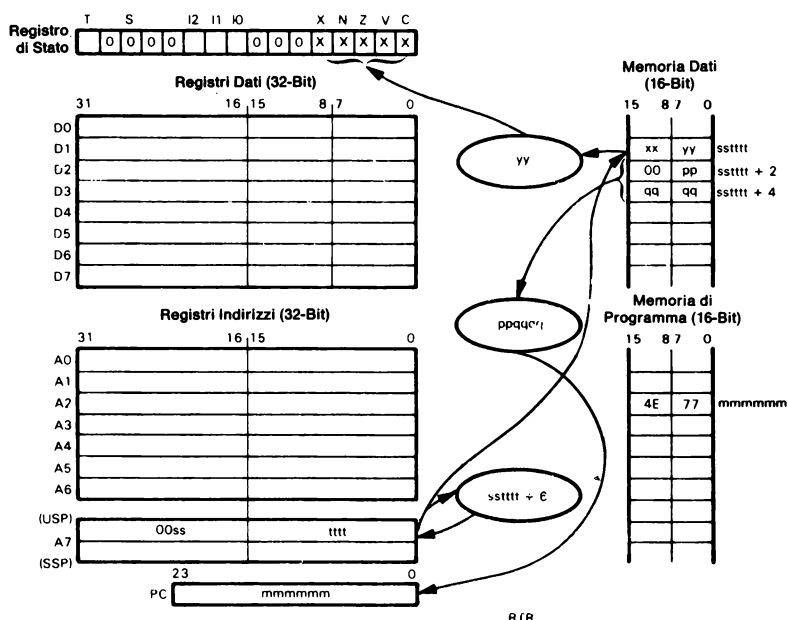
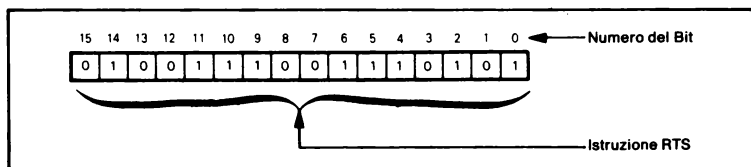



Figura 22-46. Esecuzione dell'Istruzione RTR.

RTS (Ritorno da una Subroutine)

Questa istruzione fa sì che il controllo ritorni da una subroutine, al programma chiamante prelevando l'indirizzo di ritorno dallo stack e mettendolo nel contatore di programma.

Il codice oggetto dell'istruzione RTS è:

La Figura 22-47 illustra l'esecuzione dell'istruzione RTS. Il vecchio contenuto del contatore di programma va perso. Dopo aver prelevato ciascuna word, il processore incrementa di 2 il puntatore allo stack, per cui il valore finale del puntatore è maggiore di quattro rispetto a quello iniziale. Ogni subroutine contiene normalmente almeno un'istruzione RTS; è l'ultima istruzione della subroutine ad essere eseguita ed è quella che restituisce il controllo al programma chiamante. RTS non modifica i flag di stato.

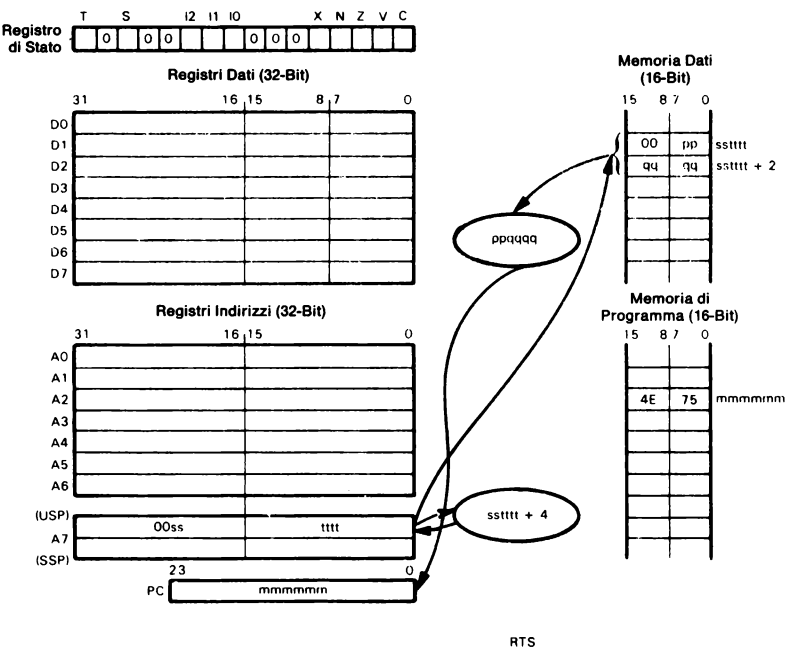
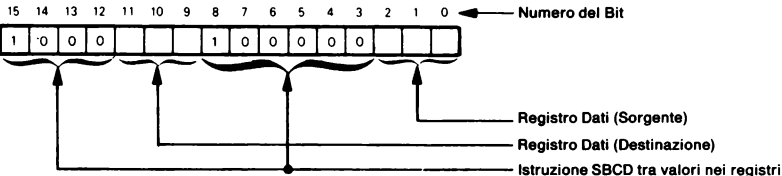
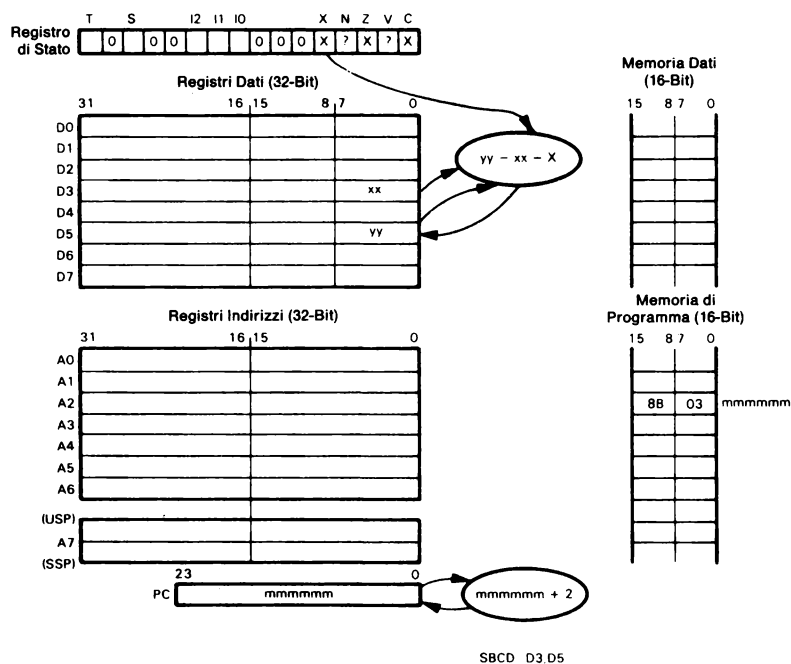
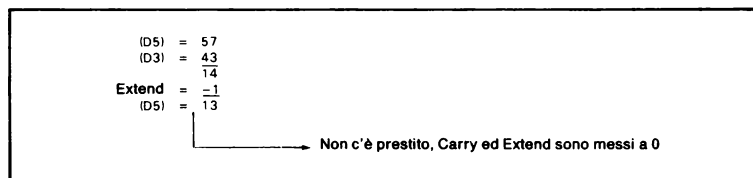



Figura 22-47. Esecuzione dell'Istruzione RTS.

SBCD (Sottrazione Decimale con Extend tra Valori nei Registri)

Questa istruzione sottrae il contenuto del registro dati sorgente ed il valore del flag di Extend (X) dal contenuto del registro dati destinazione. Il risultato è memorizzato nella locazione di destinazione. La sottrazione viene eseguita usando l'aritmetica decimale codificata binaria (BCD) e sono coinvolti soltanto gli otto bit meno significativi del registro dati.

Il codice oggetto di questa istruzione è:

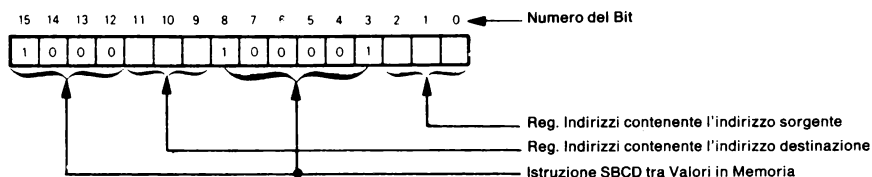
Figura 22-48. Esecuzione dell'Istruzione SBCD con gli Operandi nei Registri Dati.

La Figura 22-48 mostra l'esecuzione dell'istruzione SBCD, con il registro D3 come registro sorgente e D5 come registro destinazione. Supponiamo che xx = 43_{10}, Extend = 1 e yy = 57_{10}. Dopo che il processore avrà eseguito l'istruzione SBCD D3,D5, il contenuto di D5 sarà 13_{10}.

Questa istruzione non modifica i bit dei registri dati compresi tra 8 e 31.

I flag di Carry (C) e di Extend (X) sono posti a 1 se si verifica un prestito in seguito alla sottrazione, altrimenti sono azzerati. Il flag di Zero (Z) è azzerato, in caso di risultato diverso da zero, altrimenti resta invariato. I flag N e V non assumono valori definiti.

Si noti che il flag di Zero non è modificato se il risultato è uguale a zero. Con l'aritmetica in precisione multipla, per prima cosa bisogna porre a uno il flag di Zero (usando MOVE a CCR) e, quindi, eseguire un'operazione. Se una qualsiasi parte del risultato è diversa da zero, il flag Z sarà azzerato; altrimenti il risultato è zero ed il flag Z rimane a 1.

SBCD (Sottrazione Decimale con Extend tra Valori in Memoria)

Questa istruzione sottrae il contenuto della locazione di memoria sorgente ed il valore del flag di Extend (X) dal contenuto della locazione di memoria destinazione. Il risultato viene salvato nella locazione di destinazione. L'indirizzo dell'operando sorgente è in un registro indirizzi e quello dell'operando destinazione in un altro registro indirizzi. Il contenuto di entrambi questi registri viene decrementato prima dell'operazione. La sottrazione viene eseguita utilizzando l'aritmetica decimale codificata binaria (BCD).

Il codice oggetto per questa forma dell'istruzione SBCD è:

La Figura 22-49 mostra l'esecuzione dell'istruzione SBCD, con il registro A1 che contiene l'indirizzo dell'operando sorgente ed il registro A4 quello dell'operando destinazione. Come potete notare, il contenuto di entrambi i registri indirizzi viene decrementato prima

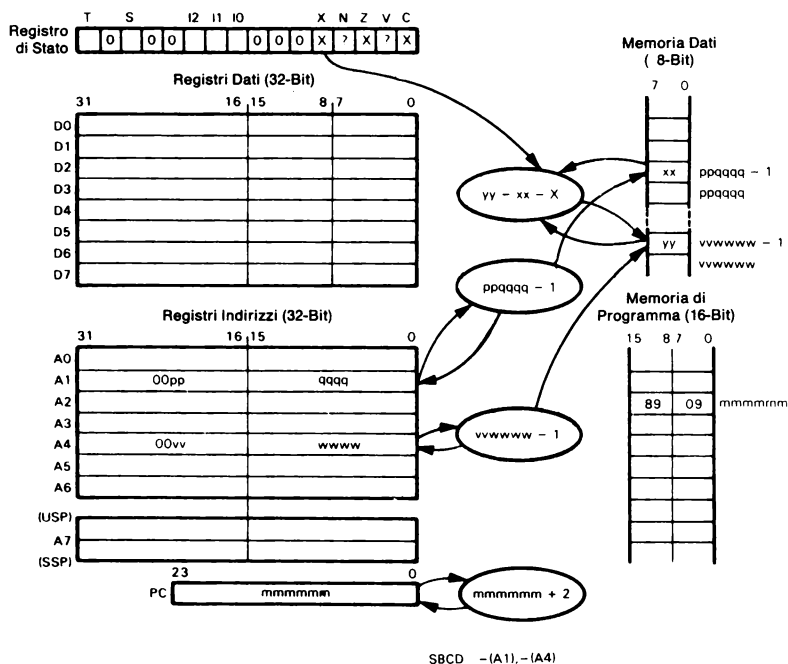



Figura 22-49. Esecuzione dell'Istruzione SBCD con Operandi in Memoria.

che gli operandi siano utilizzati per l'operazione SBCD. Questo facilita la sottrazione BCD multibyte, dal momento che una stringa di cifre BCD che rappresenta un numero decimale è normalmente memorizzata con le cifre meno significative negli indirizzi di memoria più alti. Una descrizione della sottrazione multibyte BCD la trovate nel Capitolo 8.

Gli effetti di questa versione dell'istruzione SBCD sui flag di stato sono gli stessi descritti per la versione a registro.

Scc (Set di un Byte in Base ad una Condizione)

Questa istruzione controlla lo stato di un determinato codice di condizione (cc). Se la condizione specificata è soddisfatta, allora i bit di un determinato byte dell'operando destinazione sono tutti posti a uno.

Se la condizione non è soddisfatta il byte è azzerato completamente. I tipi di indirizzamento utilizzabili per indicare l'operando destinazione sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione S_{cc} è:

I bit 8-11 della word d'istruzione indicano il codice di condizione che deve essere controllato. **La Tabella 22-3 elenca le condizioni che possono essere utilizzate con questa istruzione** ed indica quali flag del registro di stato servono per stabilire se il test ha avuto esito positivo.

Tabella 22-3. Condizioni utilizzabili con Scc.

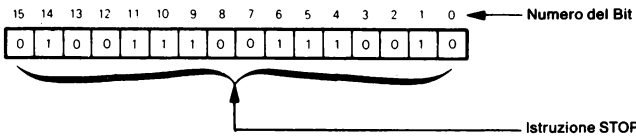
Mnemonico (cc)	Condizione	Campo Condizione	Test
T	Vero	0000	1
F	Falso	0001	0
HI	Alto	0010	$\overline{C} \wedge \overline{Z}$
LS	Basso o Uguale	0011	$C \vee Z$
CC	Carry=0	0100	$\overline{C}$
CS	Carry=1	0101	C
NE	Diverso	0110	$\overline{Z}$
EQ	Uguale	0111	Z
VC	Non Overflow	1000	$\overline{V}$
VS	Overflow	1001	V
PL	Più	1010	$\overline{N}$
MI	Meno	1001	N
GE	Maggiore o Uguale	1100	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
LT	Minore	1101	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
GT	Maggiore	1110	$(N \wedge V \wedge \overline{Z}) \vee (\overline{N} \wedge V \wedge \overline{Z})$
LE	Minore o Uguale	1111	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$

La Figura 22-50 mostra l'esecuzione di un'istruzione SPL, con indirizzamento indiretto a registro indirizzi per specificare il byte da modificare. La condizione specificata (PL) è vera, se il flag di Negativo (N) è zero. In questo caso, il byte di memoria specificato conterrà degli uno in tutte le posizioni. Se N = 1, la condizione non è soddisfatta ed il byte di memoria specificato sarà azzerato.

Nessuno dei flag di stato viene modificato.

STOP (Caricamento del Registro di Stato e Stop)

Questa istruzione carica nel registro di stato un valore immediato a 16 bit, contenuto nella memoria di programma, ed il processore non preleva, nè esegue altre istruzioni. L'esecuzione delle istruzioni non riprende finchè non si verfica un'Exception di tipo Trace, Interrupt o Reset.

Il codice oggetto dell'istruzione STOP è:

I 16 bit di dato immediato successivi alla word d'istruzione vengono messi nell'intero registro di stato. Il contenuto del contatore di programma è aumentato di quattro, in modo da indicare l'istruzione successiva. Il processore, quindi, attende che si verifichi un'Exception di tipo Trace, Interrupt o Reset.

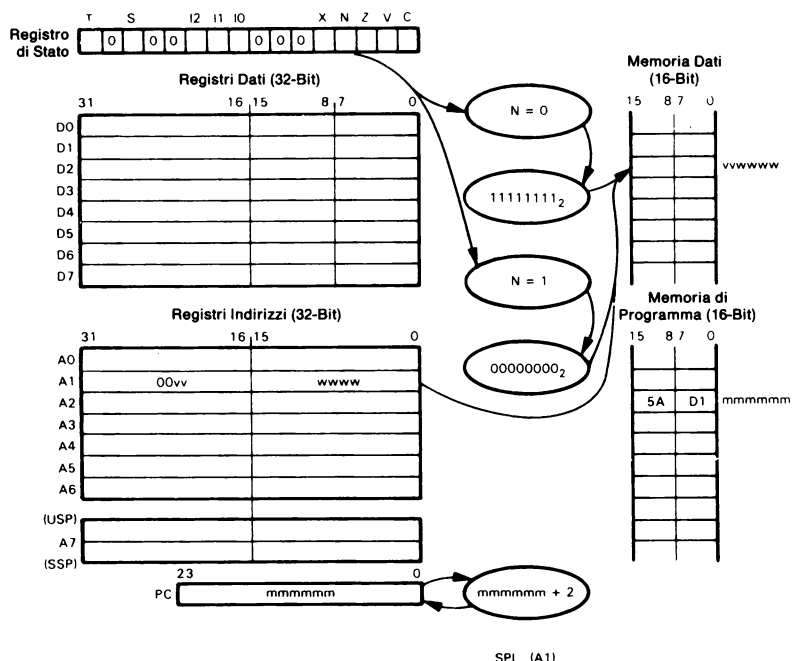



Figura 22-50. Esecuzione dell'Istruzione SPL con Indirizzamento Indiretto a Registro Indirizzi.

Un'Exception di tipo Trace si verifica immediatamente, se il modo Trace è attivo (il bit T del registro di stato = 1) nel momento in cui si verifica l'istruzione STOP.

Un'Exception da Interrupt si verifica nel caso che venga rilevata una richiesta di interrupt con priorità più elevata, rispetto al valore della maschera di interrupt del registro di stato.

Se il segnale della linea di RESET è messo basso, si verifica un'Exception da Reset, che causa la fine dell'istruzione STOP.

L'istruzione STOP è un'istruzione privilegiata e può essere eseguita soltanto quando il processore è in modo Supervisore. Se cercate di eseguire questa istruzione dal modo Utente, provocherete un'Exception dovuta a violazione di privilegio.

SUB (Sottrazione Binaria)

Questa istruzione sottrae il contenuto dell'operando sorgente dall'operando destinazione e salva il risultato nella locazione di destinazione.

Esistono due forme generali per questa istruzione. Nella prima, uno degli operandi è rappresentato da un registro dati, che conterrà anche il risultato finale. In questo caso, sono consentiti tutti i tipi di indirizzamento:

	Operando		Campo Ind.Eff. Sorgente	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi	X*		001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

* Non è consentito con operazioni della grandezza di un byte

Riguardo agli indirizzamenti, la sola limitazione sta nel fatto che il tipo diretto a registro indirizzi non può fornire l'operando sorgente se la grandezza di questo è di un byte, in quanto i registri indirizzi non sono in grado di gestire dati di un byte.

Nell'altra forma è sempre un registro dati a fornire l'operando sorgente, mentre l'operando destinazione può essere specificato mediante uno dei seguenti modi di indirizzamento:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati	X			
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto per l'istruzione SUB è:

Figura 22-51. Esecuzione dell'Istruzione SUB con Indirizzamento Indiretto a Registro Indirizzi.

La Figura 22-51 illustra l'esecuzione dell'istruzione SUB, con l'impiego dell'indirizzamento indiretto a registro indirizzi; il registro A2 fornisce l'indirizzo dell'operando sorgente posto in memoria ed il registro D3 serve come registro destinazione. Il contenuto della locazione di memoria wwvvvv viene sottratto dal contenuto del registro D3 ed il risultato viene salvato nel registro D3.

I flag di Carry (C) e di Extend (X) sono posti a 1 se in seguito all'operazione di sottrazione si verifica un prestito; altrimenti sono azzerati. Il flag di Zero (Z) diventa uno se il risultato è zero ed è azzerato in caso contrario. Il flag di Negativo (N) è posto a uno se il risultato è negativo, altrimenti viene azzerato. Il flag di Overflow (V) diventa uno in presenza di overflow, altrimenti anch'esso viene azzerato. Nel Capitolo 8 troverete una descrizione dell'interazione di questi flag nelle operazioni su numeri privi di segno e in complemento a due.

SUBA (Sottrazione con un Valore e il Risultato in un Registro Indirizzi)

Questa istruzione è un caso speciale dell'istruzione SUB e sottrae un operando sorgente da un registro indirizzi destinazione. Il risultato è salvato nel registro indirizzi destinazione.

Sono consentiti tutti i modi di indirizzamento:

Modi di Indirizzamento Possibili	Operando		Campo Ind.Eff. Sorgente	
	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati	X		000	rrr
Diretto a Registro Indirizzi	X	X	001	rrr
Indiretto a Registro Indirizzi	X		010	rrr
Indiretto a Registro con Postincremento	X		011	rrr
Indiretto a Registro con Predecremento	X		100	rrr
Indiretto a Registro con Spostamento	X		101	rrr
Indiretto a Registro con Indice	X		110	rrr
Assoluto Corto	X		111	000
Assoluto Lungo	X		111	001
Relativo al Contatore di Programma con Spostamento	X		111	010
Relativo al Contatore di Programma con Indice	X		111	011
Immediato	X		111	100

Il codice oggetto dell'istruzione SUBA è:

Specificando come operando sorgente una word a 16 bit, essa viene trasformata in una long word mediante estensione del segno e l'operazione SUBA è eseguita sul registro indirizzi specificato usando tutti i suoi 32 bit.

Una differenza sostanziale rispetto all'istruzione SUB standard è che l'istruzione SUBA non modifica nessuno dei flag di stato.

SUBI (Sottrazione di un Dato Immediato)

Questa istruzione sottrae un dato immediato, presente nelle locazioni della memoria di progamma successive, dall'operando destinazione. Il risultato viene salvato nella locazione o nel registro di destinazione.

I modi di indirizzamento utilizzabili con questa istruzione sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	x			

Il codice oggetto di questa istruzione è:

Il formato dell'operazione SUBI può essere di un byte, di una word o di una long word. Il dato immediato segue la word d'istruzione in memoria e deve corrispondere al formato specificato. Perciò, una o due word di dato immediato seguiranno il codice operativo dell'istruzione nella memoria di programma. Se l'operando deve essere di un byte, verrà utilizzato il byte di ordine basso (secondo) della word di dato immediato. L'assemblatore provvede automaticamente a selezionare il byte giusto.

Il funzionamento dell'istruzione SUBI è sostanzialmente lo stesso illustrato in precedenza per l'istruzione ADDI e, quindi, vi rimandiamo a quella istruzione ed alla relativa figura (Figura 22-5) per maggiori dettagli.

I bit di stato, X, N, Z, e C, sono modificati in modo analogo a quanto accade con l'istruzione SUB.

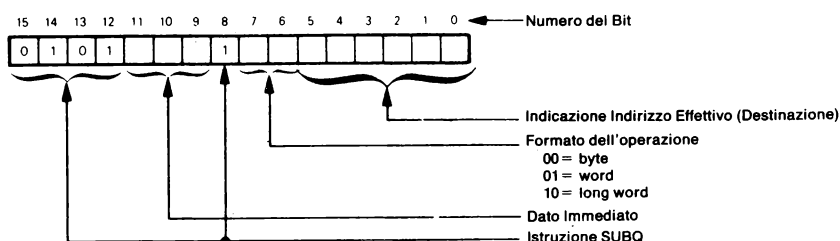
SUBQ (Sottrazione di un Dato Immediato nella Word di Codice Oggetto)

Questa istruzione sottrae il dato immediato, contenuto nella word di codice oggetto dell'istruzione, dall'operando destinazione. Il risultato viene salvato nella locazione di destinazione. I modi di indirizzamento utilizzabili sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	**Sorgente**	**Destinazione**	**Modo**	**Num. Registro**
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi		X*	001	rrr
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato	x			

* Non è consentito con operazioni della grandezza di un byte

Naturalmente, l'indirizzamento diretto a registro indirizzi non può servire per indicare l'operando destinazione se la grandezza indicata è quella di un byte.

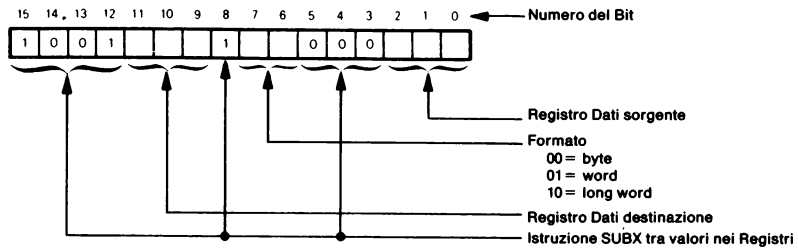
Il codice oggetto dell'istruzione SUBQ è:

I tre bit di dato immediato sono contenuti nei bit 9, 10 e 11 del codice oggetto dell'istruzione. Perciò è possibile utilizzare valori compresi tra 1 e 8. Un dato immediato costituito da tutti zeri è considerato come 8.

Il funzionamento dell'istruzione SUBQ è sostanzialmente lo stesso illustrato in precedenza per l'istruzione ADDQ e, quindi, vi rimandiamo a quella istruzione ed alla relativa figura (Figura 22-6) per maggiori dettagli.

I bit di stato, X, N, Z, e C, sono modificati in modo analogo a quanto accade con l'istruzione SUB.

SUBX (Sottrazione con Extend tra Valori nei Registri)

Questa istruzione sottrae il contenuto del registro dati sorgente ed il valore del flag di Extend (X) dal contenuto del registro dati destinazione. La grandezza indicata per l'operazione può essere di un byte, di una word o di una long word.

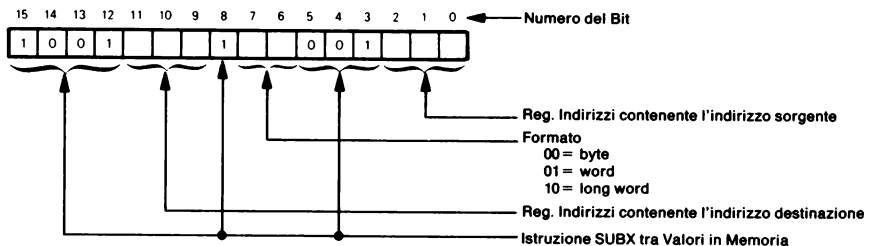
Il codice oggetto dell'istruzione è:

Il funzionamento dell'istruzione SUBX è sostanzialmente lo stesso illustrato in precedenza per l'istruzione ADDX e, quindi, vi rimandiamo a quella istruzione ed alla Figura 22-7 per una descrizione completa.

I flag di Carry (C) e di Extend (X) diventano 1 se in seguito all'operazione si è verificato un prestito; in caso contrario vengono azzerati. Il flag di Zero (Z) è azzerato nel caso di un risultato diverso da zero, altrimenti resta invariato. Il flag di Negativo (N) è posto a 1 se il risultato è zero; altrimenti viene azzerato. Il flag di Overflow (V) è posto a 1 in presenza di un overflow ed azzerato in caso contrario.

Il flag di Zero non viene modificato se il risultato è uguale a zero. Nell'aritmentica in precisione multipla, bisogna, prima di tutto, porre a uno il flag di Zero (con MOVE in CCR), quindi eseguire l'operazione. Se una parte qualsiasi del risultato è diversa da zero, il flag Z sarà azzerato; altrimenti resterà a 1, indicando che il risultato è zero.

SUBX (Sottrazione con Extend tra Valori in Memoria)

Questa istruzione sottrae, in forma binaria, il contenuto della locazione di memoria sorgente ed il valore del flag di Extend (X) dal contenuto della locazione di memoria destinazione. Il risultato viene messo nella locazione di destinazione. L'indirizzo di memoria dell'operando sorgente è contenuto in un registro indirizzi e quello dell'operando destinazione si trova in un altro registro indirizzi. il contenuto di entrambi questi registri viene decrementato prima dell'operazione.

Il codice oggetto per questa versione dell'istruzione SUBX è:

La grandezza del dato può essere di un byte, di una word o di una long word.

Il funzionamento di questa istruzione è sostanzialmente lo stesso illustrato in precedenza per l'istruzione ADDX sulla memoria e, quindi, vi rimandiamo a quella istruzione ed alla Figura 22-8 per una descrizione completa.

L'indirizzamento indiretto a registro con predecremento, impiegato con l'istruzione SUBX, facilita l'uso dell'aritmetica binaria in precisione multipla, dal momento che i registri indirizzi sono automaticamente modificati per accedere al byte, alla word o alla long word successive. Vi rimandiamo al Capitolo 8 per una trattazione completa dell'aritmetica in precisione multipla.

L'istruzione SUBX modifica i flag di stato X, N, Z e C.

SWAP (Scambio delle Word di un Registro)

Questa istruzione scambia il contenuto della word meno significativa di un registro dati con quello della word più significativa dello stesso registro.

Il codice oggetto dell'istruzione SWAP è:

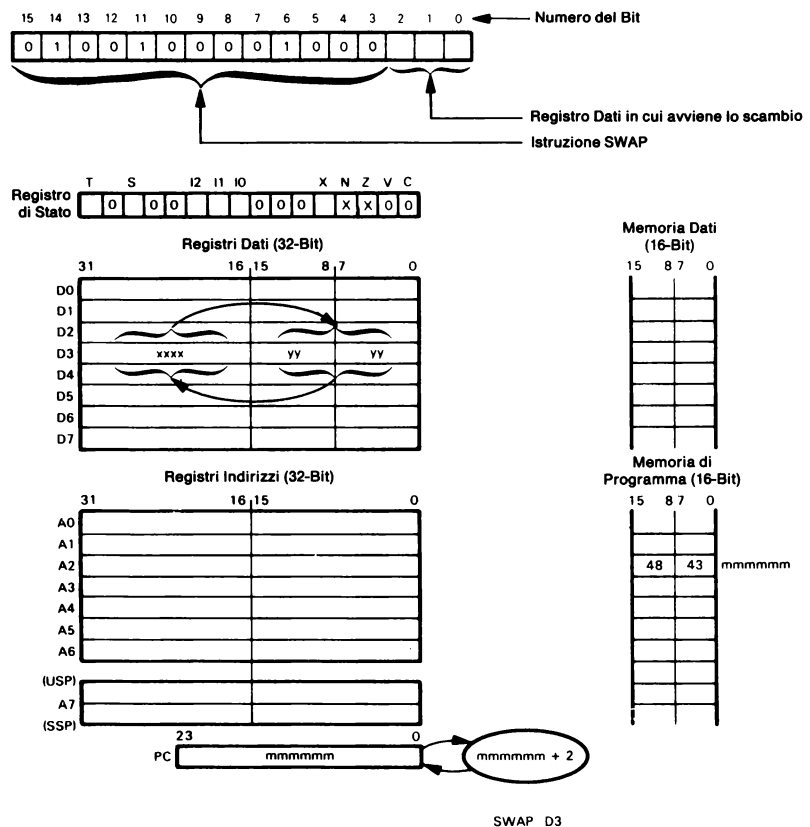



Figura 22-52. Esecuzione dell'Istruzione SWAP.

La Figura 22-52 mostra l'esecuzione di un'istruzione SWAP, con il registro D3 come operando. Una volta eseguita l'istruzione mostrata nella figura, i 16 bit meno significativi di D3 conterranno xxxx ed i 16 più significativi conterranno yyyy.

Il flag di Negativo (N) diventerà uno, se il nuovo valore del bit 31 è 1, altrimenti sarà azzerato. Il flag di Zero (Z) sarà posto a 1, se, avvenuto lo scambio, il risultato a 32 bit è uguale a zero; sarà azzerato in caso contrario.

I flag di Carry (C) e di Overflow (V) sono sempre azzerati. Il flag di Extend (X) resta invariato.

TAS (Test and Set Indivisibile)

Questa istruzione controlla un byte del dato presente nell'operando destinazione, mette a 1 o azzera i flag di stato N e Z, in base al risultato del test, e, quindi, pone a 1 il bit di ordine alto dell'operando destinazione. L'istruzione viene eseguita dal processore usando un solo ciclo di lettura-modifica-scrittura della memoria ed è, perciò, indivisibile; non può subire interrupt e nessun altro processore o dispositivo esterno può accedere all'operando durante l'esecuzione.

I tipi di indirizzamento che possono essere usati per indicare il byte da controllare sono i seguenti:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione TAS è:

L'istruzione TAS fornisce a dei programmi eseguiti separatamente, un mezzo per sincronizzare le loro attività: ad esempio per sincroniz-

zare l'accesso a dati comuni. Con TAS è possibile controllare un flag per verificare se un altro programma lo ha posto a uno e contemporaneamente metterlo a 1. Senza un'istruzione di questo tipo il vostro programma potrebbe controllare un flag rilevare che il suo valore è zero e quindi subire un interrupt prima di averlo posto a 1. Il programma che ha provocato l'interrupt potrebbe a sua volta controllare e porre a 1 quello stesso flag. Dopo l'interrupt, il vostro programma continuerebbe, come se quel flag fosse ancora a zero. Perciò le richieste di interrupt o di bus provenienti da dispositivi esterni non vengono soddisfatte durante l'esecuzione di un'istruzione TAS, grazie al fatto che viene utilizzato un solo ciclo di lettura-modifica-scrittura. Questo garantisce che l'intera istruzione verrà eseguita senza che l'operando destinazione sia nel frattempo, modificato.

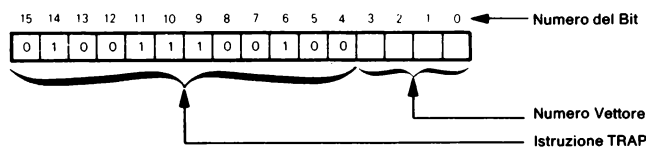
Il flag di Negativo (N) diventa uno se il bit più significativo dell'operando è 1, altrimenti viene azzerato. Si noti che questo test si verifica all'inizio dell'istruzione (dopo la parte del ciclo di memoria riservata alla lettura); il bit più significativo sarà posto a 1 durante la parte del ciclo riservata alla scrittura.

Il flag Z diventerà 1, se il contenuto dell'operando è uguale a zero; sarà azzerato in caso contario. I flag di Overflow (V) e di Carry (C) sono sempre azzerati. Il flag di Extend (X) resta invariato.

TRAP (Trap)

Questa istruzione provoca un'Exception. Il valore del contatore di programma, incrementato in modo da indicare l'istruzione successiva, viene messo sullo stack di sistema; quindi anche la word del registro di stato viene messa sullo stack. Dalla opportuna locazione della tabella dei vettori di Exception sono poi prelevate e poste nel contatore di programma due word. L'esecuzione continuerà a partire dalla locazione indicata dal nuovo valore presente nel contatore di programma.

L'istruzione TRAP mette i dati allo stack Supervisore; questo significa che viene usato sempre il puntatore allo stack Supervisore (SSP), indipendentemnte dal fatto che l'istruzione TRAP sia iniziata nel modo Supervisore o in quello Utente.

Il codice oggetto dell'istruzione TRAP è:

Con l'istruzione TRAP è possibile specificare sedici vettori diversi. La Tabella 22-4 mostra l'intera tabella dei vettori di Exception.

In questa tabella gli indirizzi dei vettori dell'istruzione TRAP occupano le posizioni dalla 32 alla 47. Ciascun indirizzo consiste di

due word (quattro byte) che saranno caricate nel contatore di programma. Perciò l'indirizzo del vettore relativo alla TRAP #0 occupa le locazioni di memoria $080\text{-}083_{16}$; il vettore per la TRAP #15 occupa le locazioni $0BC\text{-}0BF_{16}$.

Tabella 22-4. Tabella dei Vettori di Exception.

Numero Vettore	Indirizzo		Assegnamento
	Dec	Hex	
0	0	000	Reset: SSP Iniziale
	4	004	Reset: PC Iniziale
2	8	008	Errore di Bus
3	12	00C	Errore di Indirizzo
4	16	010	Istruzione Illegale
5	20	014	Divisione per 0
6	24	018	Istruzione CHK
7	28	01C	Istruzione TRAPV
8	32	020	Violazione di Privilegio
9	36	024	Trace
10	40	028	Emulazione Line 1010
11	44	02C	Emulazione Line 1111
12*	48	030	(Non assegnato, riservato)
13*	52	034	(Non assegnato, riservato)
14*	56	038	(Non assegnato, riservato)
15*	60	03C	(Non assegnato, riservato)
16-23*	64	040	(Non assegnato, riservato)
	95	05F	—
24	96	060	Interrupt Spurio
25	100	064	Autovettore Interrupt Livello 1
26	104	068	Autovettore Interrupt Livello 2
27	108	06C	Autovettore Interrupt Livello 3
28	112	070	Autovettore Interrupt Livello 4
29	116	074	Autovettore Interrupt Livello 5
30	120	078	Autovettore Interrupt Livello 6
31	124	07C	Autovettore Interrupt Livello 7
32	128	080	Istruzione TRAP #0
33	132	084	Vettore Istruzione TRAP #1
34	136	088	Vettore Istruzione TRAP #2
45	18C	0B4	Vettore Istruzione TRAP #13
46	184	0B8	Vettore Istruzione TRAP #14
47	188	0BC	Vettore Istruzione TRAP #15
48-63*	192	0C0	(Non assegnato, riservato)
	255	0FF	—
64-255	256	100	Vettori Interrupt Utente
	1023	3FF	—

* Riservato alla Motorola. Per conservare la compatibilità con i sistemi della Motorola, evitare l'uso di queste locazioni

La Figura 22-53 mostra l'esecuzione di un'istruzione TRAP. Si tratta di una TRAP #1 che accede ai quattro byte con l'indirizzo del vettore a partire dalla locazione di memoria 084_{16}. Il contenuto di queste due word (00ppqqqq) viene caricato nel contatore di programma dopo che il contenuto precedente è stato salvato allo stack di Sistema, insieme al registro di stato.

Dopo aver salvato il contenuto del registro di stato, il flag di Trace (T) è posto a 0 ed il flag Supervisore (S) diventa 1. Nessun altro bit del registro di stato viene modificato.

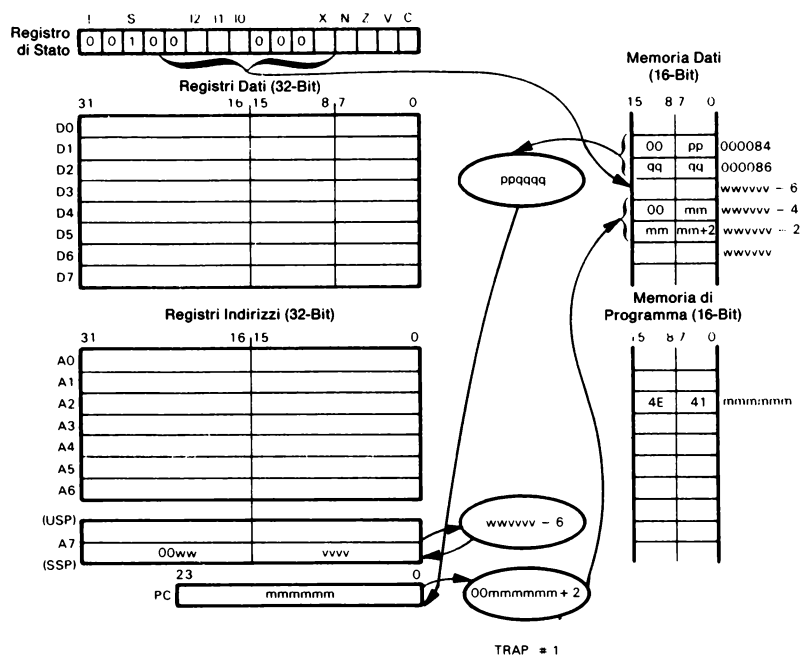



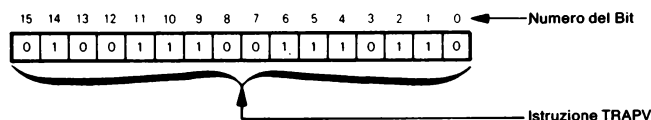
Figura 22-53. Esecuzione dell'Istruzione TRAP.

La sequenza illustrata nella Figura 22-53 viene eseguita anche quando le TRAP sono iniziate automaticamente dal processore, in seguito al tentativo di effettuare una divisione per zero o di eseguire un codice operativo illegale o non implementato. Questi altri tipi di Exception utilizzano altri vettori elencati sempre nella Tabella 22-4.

Per una completa descrizione delle Trap e degli altri tipi di Exception vi rimandiamo al Capitolo 15.

TRAPV (Trap per un Overflow)

Questa istruzione causa un'Exception se il bit di Overflow (V) del registro di stato è 1 nel momento in cui viene eseguita l'istruzione. La sequenza è analoga a quella illustrata nella Figura 22-53, tranne per il fatto che il vettore utilizzato è il numero 7 (quello relativo all'istruzione TRAPV), che ritroviamo a partire dalla locazione $01C_{16}$ (cfr. Tabella 22-4).

Il codice oggetto dell'istruzione TRAPV è:

TST (Test di un Operando)

Questa istruzione pone a 1 o azzera i flag di Negativo (N) e di

Zero (Z) in base al contenuto dell'operando destinazione. I tipi di indirizzamento utilizzabili per indicare l'operando da controllare sono:

	Operando		Campo Ind.Eff. Destinazione	
Modi di Indirizzamento Possibili	Sorgente	Destinazione	Modo	Num. Registro
Diretto a Registro Dati		X	000	rrr
Diretto a Registro Indirizzi				
Indiretto a Registro Indirizzi		X	010	rrr
Indiretto a Registro con Postincremento		X	011	rrr
Indiretto a Registro con Predecremento		X	100	rrr
Indiretto a Registro con Spostamento		X	101	rrr
Indiretto a Registro con Indice		X	110	rrr
Assoluto Corto		X	111	000
Assoluto Lungo		X	111	001
Relativo al Contatore di Programma con Spostamento				
Relativo al Contatore di Programma con Indice				
Immediato				

Il codice oggetto dell'istruzione TST è:

L'operando da controllare può essere un byte, una word oppure una long word.

La Figura 22-54 illustra l'esecuzione dell'istruzione TST, con indirizzamento indiretto a registro indirizzi. Il registro A4 è usato per indicare la word di memoria di cui bisogna controllare il contenuto. Supponiamo che xxxx = 0000_{16}. Una volta eseguita l'istruzione TST (A3) i flag di Carry (C) e di Overflow (V) conterranno zero, il flag di Negativo (N) sarà azzerato ed il flag Z sarà 1. L'istruzione TST non modifica il contenuto di nessun registro e di nessuna locazione di memoria.

L'istruzione TST permette al programmatore di azzerare o porre a 1 i flag del registro di stato in base al contenuto di una locazione di memoria o di un registro senza eseguire nessuna operazione e senza cambiare il contenuto di alcun registro o locazione di memoria.

Il flag di Negativo (N) diventa 1 se l'operando controllato è negativo, altrimenti viene azzerato. Il flag Z è posto a 1 se l'operando è zero; sarà azzerato in caso contrario. I flag di Carry (C) e di Overflow (V) sono sempre azzerati. Il flag di Extend (X) resta invariato.

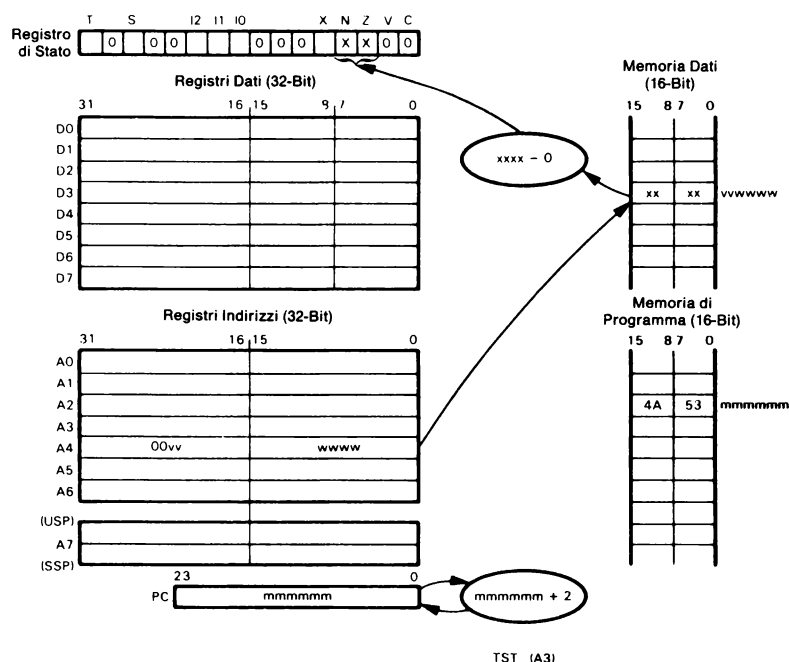
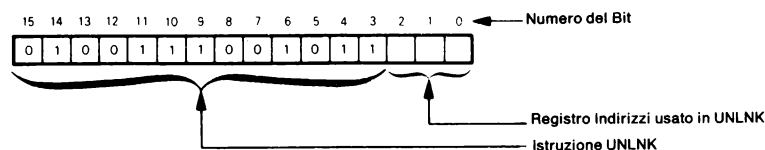



Figura 22-54. Esecuzione dell'Istruzione TST con Indirizzamento Indiretto a Registro Indirizzi.

UNLK (Unlink)

Questa istruzione carica nel puntatore allo stack di sistema il contenuto di un determinato registro indirizzi (un "puntatore di frame") nel quale, a sua volta, viene messa una long word prelevata dalla sommità dello stack. Perciò, sia al puntatore di frame che al puntatore allo stack di Sistema saranno restituiti i valori che avevano prima dell'esecuzione di un'istruzione LINK.

Il codice oggetto dell'istruzione UNLK è:

La Figura 22-55 mostra l'esecuzione dell'istruzione UNLK con il registro A2 come puntatore di frame. L'istruzione UNLK viene usata per ripulire lo stack alla fine di una subroutine iniziata con un'istruzione LINK. Per maggiori dettagli sulle istruzioni LINK ed UNLK, vi rimandiamo alla descrizione dell'istruzione LINK.

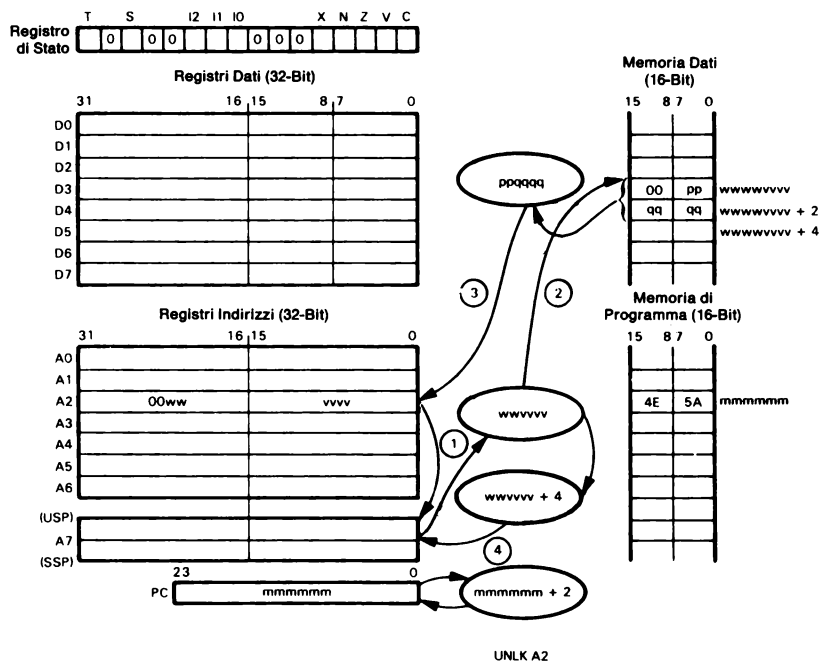



Figura 22-55. Esecuzione dell'Istruzione UNLK.

L'istruzione UNLK non modifica nessuno dei flag del registro di stato.

SEZIONE VI

APPENDICE

La parte seguente del volume contiene una serie di tabelle di riferimento per il set d'istruzioni dell'MC68000.

L'Appendice A riassume il funzionamento e gli effetti delle istruzione dell'MC68000 ed è organizzata in base alle diverse funzioni, in modo da evidenziare le possibilità di questo microprocessore. L'Appendice B mostra il codice oggetto di tutte le istruzioni (elencate in ordine alfabetico in base al corrispondente mnemonico) ed il relativo tempo di esecuzione in cicli di clock. L'Appendice B può essere di aiuto nell'assemblaggio manuale delle istruzioni dell'MC68000. L'Appendice C elenca tutti i codici oggetto validi ed i corrispondenti mnemonici in ordine numerico e può essere utilizzata per il controllo manuale ed il disassemblaggio di un programma oggetto, due tecniche adottate molto spesso in fase di debugging.

APPENDICE A

IL SET DI ISTRUZIONI

La Tabella A-2 riassume il set di istruzioni dell'MC68000. La colonna MNEMONICI contiene i mnemonici delle istruzioni (ad es. MOVE, ADD, JMP). La colonna OPERANDI elenca i corrispondenti operandi.

La parte fissa di un'istruzione in linguaggio assembly è indicata in LETERE MAIUSCOLE. La parte destinata a variare (num. del registro, indirizzo, dato immediato, ecc.) è indicata in caratteri minuscoli.

Anche in questa tabella sono riportati i BYTE ed i CICLI DI CLOCK richiesti da ciascuna istruzione. Consultate l' Appendice B per la relativa descrizione.

MNEMONICI ALTERNATIVI

Il set di istruzioni dell'MC68000 permette una vasta scelta di mnemonici. Una "I" aggiunta in coda ad un mnemonico indica un'operazione immediata. Aggiungendo, invece, una "A" si identificherà un'operazione relativa ad un registro indirizzi. Una ".S" indica la forma breve di un'istruzione di salto condizionato.

La scelta dei mnemonici è riassunta nella Tabella A-1 sotto le intestazioni seguenti:

Mnemonici Primari	Elenco dei Mnemonici in forma nominale.
Mnemonici Alternativi	Elenco dei mnemonici utilizzabili in alternativa a quelli primari
Operando	Mostra la categoria dell'operando consentito con i mnemonici primari ed alternativi. XX indica la possibilità di utilizzare un operando qualsiasi.
Descrizione	Identifica l'operazione.

Per semplicità, le tabelle seguenti riportano solo i mnemonici primari.

Non esistono mnemonici alternativi per le variazioni di tipo X (Extend), M (Multiple) e P (Peripheral Data). Non è possibile omettere questi suffissi dai rispettivi mnemonici.

Ricordate che l'assemblatore sceglierà la versione "Quick" di un'istruzione, ogni volta che ciò sarà possibile (ad es. MOVEQ,

ADDQ, SUBQ). Perciò, anche utilizzando i mnemoci nella forma più generale (MOVE, ADD e SUB) otterrete sempre lo stesso codice oggetto.

Ad esempio: MOVE.L #40,D2
è codificato come: MOVEQ #40, D2

Un altro esempio: ADD #1,D0
è codificato come: ADDQ #1,D0

STATO

Gli effetti dell'esecuzione di un'istruzione sui bit di stato sono riportati nella Tabella A-2. I bit di stato sono:

T – modo Trace
S – modo (o stato) Supervisore
X – bit di Extend
N – bit di Negativo (o Segno)
Z – bit di Zero
V – bit di Overflow
C – bit di Carry

Nella colonna STATO sono utilizzati i simboli seguenti:

X – il flag è modificato dall'esecuzione dell'istruzione
(spazio) – il flag non è modificato dall'esecuzione dell'istruzione
1 – il flag è posto a 1 dall'esecuzione dell'istruzione
0 – il flag è posto a 0 dall'esecuzione dell'istruzione

OPERAZIONE ESEGUITA

Questa colonna mostra la sequenza di operazioni che si verificano quando viene eseguita un'istruzione. (non è indicato il prelievo di un'istruzione, nè il relativo incremento del contatore di programma). Ciascuna operazione è indicata generalmente nella forma seguente

destinazione <----- sorgente

che indica la sostituzione del contenuto della destinazione con quello della sorgente. Ad esempio, l'istruzione LEA sarà rappresentata in questo modo:

[An] <--- jadr

L'indirizzo effettivo, espresso in una delle forme possibili per jadr, viene posto nel registro indirizzi specificato.

ABBREVIAZIONI

Queste sono le abbreviazioni usate per i formati delle istruzioni e la descrizione delle operazioni.

addr	Indirizzo diretto (16 0 32 bit)
An	Registro indirizzi, n = 0-7 (8, 16 o 32 bit, a seconda della dimensione dell'istruzione)
bitb	Posizione di un bit all'interno di un byte
bitl	Posizione di un bit in una long word
cc	Codici di condizione:

CC	Carry = 0	0100
CS	Carry = 1	0101
EQ	Uguale	0111
F	Falso	0001
GE	Maggiore o uguale	1100
GT	Maggiore di	1110
HI	Alto	0010
LE	Minore di o uguale	1111
LS	Basso o uguale	0011
LT	Minore di	1101
MI	Negativo	1011
NE	Diverso	0110
PL	Positivo	1010
T	Vero	0000
VC	Non Overflow	1000
VS	Overflow	1001

CCR	**Registro dei Codici di Condizione: il byte di ordine basso del registro di Stato**
count	Contatore di Shift (1-8)
dadr	Indirizzo destinazione, in uno dei seguenti modi di indirizzamento: (An) Indiretto a Registro (An)+ Indiretto a Registro con postincremento -(An) Indiretto a Registro con predecremento d16(An) Indiretto a Registro con spostamento d8(An,i) Indiretto a Registro indicizzato addr Diretto
dAn	Registro Indirizzi Destinazione. Questa forma è usata solo in presenza di due operandi An.
aDn	Registro Dati Destinazione. Questa forma è usata solo in presenza di due operandi Dn.
data3	3 bit di dato immediato
data8	8 bit di dato immediato
data16	16 bit di datto immediato
data32	32 bit di dato immediato

Dn — Registro Dati, n = 0-7 (8, 16 o 32 bit, a seconda del formato dell'istruzione)

d8 — offset di indirizzo ad 8 bit. Necessario, anche se zero, nelle istruzioni indicizzate.

d16 — offset di indirizzo a 16 bit.

i — indice

jadr — Indirizzo di salto: come sadr, ma senza (An)+ e -(An)

label — Label (etichetta) di indirizzo

madr — Indirizzo per istruzioni multiple: come dadr, ma senza (An)+ e -(An)

reg-list — Elenco con uno o più registri, ciascuno dei quali separato da una virgola. Gli elementi possono essere dei seguenti tipi:

Dn	Singolo registro dati
An	Singolo registro indirizzi
rn_1-rn	Gamma di registri

rd — Registri Destinazione (dDn o dAn)

rs — Registri Sorgente (sDn o sAn)

sadr — Indirizzo Sorgente, in uno dei seguenti tipi di indirizzamento:

(An)	Indiretto a Registro
(An)+	Indiretto a Registro con postincremento
-(An)	Indiretto a Registro con predecremento
d16(An)	Indiretto a Registro con spostamento
d8(An,i)	Indiretto a Registro indicizzato
addr	Diretto
label	Relativo al programma
label (i)	Relativo al programma indicizzato

sAn — Registro Indirizzi Sorgente. Questa forma è usata solo in presenza di due operandi An.

sDn — Registro Dati Sorgente. Questa forma è usata solo in presenza di due operandi Dn.

SR — Registro di Stato (16 bit)

USP — Puntatore allo Stack Utente. Notate che si tratta del registro A7.

vector — Vettore indirizzo di Trap: la locazione di memoria contenente l'indirizzo della routine di Trap

[[ ]] — Il contenuto della locazione di memoria puntata dal registro specificato (indirizzamento indiretto della memoria o indirizzamento implicito)

[ ] — Il contenuto di un registro o di una locazione di memoria (indirizzamento a registro o indirizzamento diretto della memoria).

Ad esempio:
[Dn] ← [[An]]
significa che il contenuto della locazione di memoria indirizzata dal Registro An viene caricato in Dn, mentre:
[Dn] ← [An]

significa che il contenuto dello stesso Registro An viene caricato in Dn.

$\bar{x}$ — Complemento del valore di x

x<y-z> — I bit da y a z di x. Ad esempio, Dn<0-7> indica il byte di ordine basso di Dn. Se il termine z è omesso, viene indicato solo il bit y. Perciò, Dn<0> è il bit meno significativo di Dn.

$+$ — Somma

$-$ — Sottrazione

$\times$ — Moltiplicazione

$\div$ — Divisione

$\wedge$ — AND Logico

$\vee$ — OR Logico

$\veebar$ — OR Esclusivo Logico

$=$ — Uguale

$\leftarrow$ — Movimento dei dati nella direzione della freccia

$\leftrightarrow$ — Scambio di dati tra due locazioni

Tabella A-1. Mnemonici Alternativi

Mnemonici Primari	Mnemonici Alternativi	Operando	Descrizione
ADD.B	ADDI.B	dato8,xx	Somma Immediata su un Byte
ADD.W	ADD	xx,xx	Somma su una Word
	ADDA.W	xx,An	Somma (su una Word) col Registro indirizzi
	ADDI.W	dato16,xx	Somma Immediata su una Word
ADD.L	ADDA.L	xx,An	Somma (su una Long Word) col Registro Indirizzi
	ADDI.L	dato32,xx	Somma Immediata su una Long Word
ADDQ.B	ADD.B	dato3,xx	Somma Veloce su un Byte
ADDQ.W	ADD	dato3,xx	Somma Veloce su una Word
	ADD.W		
ADDQ.L	ADD.L	dato3,xx	Somma Veloce su una Long Word
AND.B	ANDI.B	dato8,xx	AND Immediato su un Byte
AND.W	AND	xx,xx	AND su una Word
	ANDI.W	dato16,xx	AND Immediato su una Word
AND.L	ANDI.L	dato32,xx	AND Immediato su una Long Word
BCC	Bcc.S	xx	Diramazione Condizionata Breve
CLR.W	CLR	xx	Azzera una Word
CMP.B	CMPI.B	dato8,xx	Confronto Immediato su un Byte
CMP.W	CMP	xx,xx	Confronto su una Word
	CMPA.W	xx,An	Confronto (su una Word) con Registro Indirizzi
	CMPI.W	dato16,xx	Confronto Immediato su una Word
CMP.L	CMPA.L	xx,An	Confronto (su una Long Word) con Registro Indirizzi
	CMPI.L	dato32,xx	Confronto Immediato su una Long Word
EOR.B	EORI.B	dato8,xx	OR Esclusivo Immediato su un Byte
EOR.W	EOR	xx,xx	OR Esclusivo su una Word
	EORI.W	dato16,xx	OR Esclusivo Immediato su una Word
EOR.L	EORI.L	dato32,xx	OR Esclusivo Immediato su una Long Word
MOVE.W	MOVE	xx,xx	Trasferimento di una Word
	MOVEA.W	xx,An	Trasferimento di una Word in Registro Indirizzi
MOVE.L	MOVEA.L	xx,An	Trasferimento di una Long Word in Registro Indirizzi
MOVEQ	MOVE.L	dato32,xx	Trasferimento Veloce (sempre di Long Word)
OR.B	ORI.B	dato8,xx	OR Immediato su un Byte
OR.W	OR	xx,xx	OR su una Word
	ORI.W	dato16,xx	OR Immediato su una Word
OR.L	ORI.L	dato32,xx	OR Immediato su una Long Word
SUB.B	SUBI.B	dato8,xx	Sottrazione Immediata su un Byte
SUB.W	SUB	xx,xx	Sottrazione su una Word
	SUBA.W	xx,An	Sottrazione (su una Word) dal Registro Indirizzi
	SUBI.W	dato16,xx	Sottrazione Immediata su una Word
SUB.L	SUBA.L	xx,An	Sottrazione (su una Long Word) dal Registro Indirizzi
	SUBI.L	dato32,xx	Sottrazione Immediata su una Long Word
SUBQ.B	SUB.B	dato3,xx	Sottrazione Veloce su un Byte
SUBQ.W	SUB	dato3,xx	Sottrazione veloce su una Word
	SUB.W		
SUBQ.L	SUB.L	dato3,xx	Sottrazione Veloce su una Long Word

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	Stato S	Stato X	Stato N	Stato Z	Stato V	Stato C	Operazione eseguita
I/O e riferimenti primari alla memoria	LEA	jadr,An	2,4, o 6	2(0/0)+								[An] ← jadr Pone l'indirizzo effettivo nel registro indirizzi specificato. La misura dell'indirizzamento è "lunga", anche se l'indirizzo può essere un byte, una word, o una long word, in funzione del successivo uso[2]
	MOVE.B	(An),Dn	2	8(2/0)				X	X	0	0	[Dn<0-7>] ← [[An]] Indirizzo indiretto a registro
		(An) +,Dn	2	8(2/0)				X	X	0	0	[Dn<0-7>] ← [[An]], [An] ← [An] + 1 Indirizzo indiretto a registro con postincremento[1]
		-(An),Dn	2	10(2/0)				X	X	0	0	[An] ← [An] - 1, [Dn<0-7>] ← [[An]] Indirizzo indiretto a registro con precedecremento[1]
		d16(An),Dn (sadr)	4	12(3/0)				X	X	0	0	[Dn<0-7>] ← [[An] + d16] Indirizzo indiretto a registro con offset
		d8(An,i),Dn (sadr)	4	14(3/0)				X	X	0	0	[Dn<0-7>] ← [[An] + d8 + [i]] Indirizzo indiretto a registro indicizzato
		addr,Dn (sadr)	4 o 6	4(1/0)+				X	X	0	0	[Dn<0-7>] ← [addr] Indirizzo diretto
		label,Dn (sadr)	4	12(3/0)				X	X	0	0	[Dn<0-7>] ← [[PC] + d16] Relativo al programma
		label(i),Dn (sadr)	4	14(3/0)								[Dn<0-7>] ← [[PC] + d8 + [i]] Relativo al programma, indicizzato
												Copia in un registro dati il byte di memoria identificato da uno dei modi di indirizzamento sopra specificati. I bit 8-31 del registro dati restano invariati
	MOVE.B	Dn(An) (dadr)	2	9(1/1)				X	X	0	0	[[An]] ← [Dn<0-7>] Indirizzo indiretto a registro
		Dn,(An)+ (dadr)	2	9(1/1)				X	X	0	0	[[An]] ← [Dn<0-7>], [An] ← [An] + 1 Indirizzo indiretto a registro con postincremento[1]
		Dn,-(An) (dadr)	2	9(1/1)				X	X	0	0	[An] ← [An] - 1, [[An]] ← [Dn<0-7>] Indirizzo indiretto a registro con predecremento
		Dn,d16(An) (dadr)	4	13(2/1)				X	X	0	0	[[An] + d16] ← [Dn<0-7>] Indirizzo indiretto a registro con offset
		Dn,d8(An, i) (dadr)	4	15(2/1)				X	X	0	0	[[An] + d8 + [i]] ← [Dn<0-7>] Indirizzo indiretto a registro indicizzato
		Dn,addr (dadr)	4 o 6	5(0/1)+				X	X	0	0	[addr] ← [Dn<0-7>] Indirizzo diretto
												Copia nella locazione identificata da uno dei modi di indirizzamento sopra descritti il byte di un registro dati
	MOVE.B	sadr,dadr	2, 4 6, 8 o 10	5(1/1)+				X	X	0	0	[dadr] ← [sadr] Copia nella locazione di memoria "destinazione" il byte di memoria "sorgente"[1]

Note:

1. Postincremento e predecremento di 1, salvo che il registro specificato sia il Puntatore allo Stack, dove l'indirizzo è variato di 2 per mantenerlo nei limiti di word.
2. L'indirizzo effettivo deve coincidere con un limite di word pari (0000, 0002, 0004...).
3. Postincremento e predecremento di 2.
4. Postincremento e predecremento di 4.
5. Per "dati immediati" e "bit immediati" si intende rispettivamente dati e bit indirizzati in modo immediato.

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
I/O e riferimenti primari alla memoria (continua)	MOVE.W	sadr, Dn	2, 4 o 6	4(1/0)+				X	X	0	0	[Dn<0-15>] ← [sadr] Copia nella word di un registro dati una word di memoria. I bit 16-31 del registro dati restano invariati[2,3]
	MOVE.W	sadr,An	2, 4 o 6	4(1/0)+								[An]<0-15>] [An <16-31>] ← [An<15>] Copia nella word di un registro indirizzi il contenuto di una word di memoria. Il segno è esteso ai bit più significativi del registro[2,3]
	MOVE.W	rs,dadr	2, 4 o 6	5(0/1)+				X	X	0	0	[dadr] ← [rs<0-15>] Copia in una word di memoria la word di un registro dati o di un registro indirizzi[2,3]
	MOVE.W	sadr,dadr	2, 4 6, 8 o 10	5(0/1)+				X	X	0	0	[dadr] ← [sadr] Copia una word da una locazione di memoria "sorgente" a una locazione di memoria "destinazione"[2,3]
	MOVE.L	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	0	0	[Dn<0-31>] ← [sadr] Copia in un registro dati una long word di memoria[2,4]
	MOVE.L	sadr,An	2, 4 o 6	8(2/0)+								[An<0-31>] ← [sadr] Copia in un registro indirizzi una long word di memoria[2,4]
	MOVE.L	rs,dadr	2, 4 o 6	10(0/2)+				X	X	0	0	[dadr] ← [rs<0-31>] Copia in una long word di memoria un registro dati o indirizzi[2,4]
	MOVE.L	sadr,dadr	2, 4 6, 8 o 10	14(1/2)+				X	X	0	0	[dadr] ← [sadr] Copia una long word da una locazione di memoria "sorgente" a una locazione "destinazione"[2,4]
	MOVEM.W	jadr,reg-list	4, 6 o 8	8 + 4n(2 + n/0)+								[reg1<0-15>] ← [[An]], [reg1<16-31>] ← [reg1<15>] [reg2<0-15>] ← [[An + 2]],[reg2<16-31>] ← [reg2<15>] [reg3<0-15>] ← [[An + 4]],[reg3<16-31>] ← [reg3<15>] . . . [regn<0-15>] ← [[An + 2n-2]],[regn<16-31>] ← [regn<15>] Copia più word di memoria consecutive nelle word dei registri specificati, nell'ordine D0-D7, A0-A7. Il segno è esteso a tutti i bit più significativi del registro[2]
	MOVEM.W	(An)+,reg-list	4	8 + 4n(2 + n/0)								[reg1<0-15>] ← [[An]],[reg1<16-31>] ← [reg1<15>],[An] ← [An + 2] [reg2<0-15>] ← [[An]],[reg2<16-31>] ← [reg2<15>],[An] ← [An + 2] [reg3<0-15>] ← [[An]],[reg3<16-31>] ← [reg3<15>], [An] ← [An + 2] . . . [regn<0-15>] ← [[An]], [regn<16-31>] ← [regn<15>],[An] ← [An + 2] Come sopra, con postincremento[3]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato							Operazione eseguita
					T	S	X	N	Z	V	C	
I/O e riferimenti primari alla memoria (continua)	MOVEM.W	reg-list,madr	4, 6 o 8	4 + 5n(1/n)+								[[An]] ← [reg1<0-15>] [[An + 2]] ← [reg2<0-15>] [[An + 4]] ← [reg3<0-15>] . . . [[An + (2n-2)] ← [regn<0-15>] Copia in più word di memoria consecutive le word dei registri specificati, nell'ordine D0-D7, A0-A7[2]
	MOVEM.W	reg-list,-(An)	4	4 + 5n(1/n)+								[An] ← [An-2],[[An]] ← [regn<0-15>] . . . [An] ← [An-2],[[An]] ← [reg3<0-15>] [An] ← [An-2],[[An]] ← [reg2<0-15>] [An] ← [An-2],[[An]] ← [reg1<0-15>] Come sopra, con predecremento[2, 3]
	MOVEM.L	jadr,reg-list	4, 6 o 8	8 + 8n(2 + 2n/0)								Come MOVEM.W, ma opera in formato long word (32 bit)[2, 4]
		(An)+,reg-list	4	8 + 8n(2 + 2n/0)								
		reg-list,madr	4, 6 o 8	4 + 10n(1/n)+								
		reg-list,-(An)	4	4 + 10n(1/n)								
	MOVEP.W	d16(An),Dn	4	16(4/0)								[Dn<8-15>] ← [[An] + d16],[An] ← [An] + 2 [Dn<0-7>] ← [[An] + d16] Copia nella word di un registro dati due byte di memoria non consecutivi. L'indirizzo è un indirizzo di byte[3]
	MOVEP.W	Dn,d16(An)	4	18(2/2)								[[An] + d16] ← [Dn<8-15>],[An] ← [An] + 2 [[An] + d16] ← [Dn<0-7>] Copia in due byte di memoria non consecutivi la word di un registro dati. L'indirizzo è un indirizzo di byte[3]
	MOVEP.L	dos(An),Dn	4	24(6/Q)								[Dn<24-31>] ← [[An] + d16],[An] ← [An] + 2 [Dn<16-23>] ← [[An] + d16],[An] ← [An] + 2 [Dn<8-15>] ← [[An] + d16],[An] ← [An] + 2 [Dn<0-7>] ← [[An] + d16] Copia quattro byte di memoria non consecutivi in un registro dati. L'indirizzo è un indirizzo di byte[3]
	MOVEP.L	Dn,d16(An)	4	28(2/4)								[[An] + d16] ← [Dn<24-31>],[An] ← [An] + 2 [[An] + d16] ← [Dn<16-23>],[An] ← [An] + 2 [[An] + d16] ← [Dn<8-15>],[An] ← [An] + 2 [[An] + d16] ← [Dn<0-7>] Copia in quattro byte di memoria non consecutivi un registro dati. L'indirizzo è un indirizzo di byte[3]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato							Operazione eseguita
					T	S	X	N	Z	V	C	
Riferimenti secondari alla memoria (operazioni su memoria)	ABCD	-(sAn),-(dAn)	2	19(3/1)			X	U	X	U	X	[sAn] ← [sAn] - 1 [dAn] ← [dAn]-1 [[dAn]] ← [[dAn]] + [[sAn]] + X Somma decimale fra due byte di memoria con riporto (bit Extend). Entrambi gli indirizzi sono indirizzi di byte[1]
	ADD.B	sadr,Dn	2, 4 o 6	4(1/0)+			X	X	X	X	X	[Dn<0-7>] ← [Dn<0-7>] + [sadr] Somma al byte di un registro dati un byte di memoria. I bit 8-31 del registro dati restano invariati[1]
	ADD.B	Dn,dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] + [Dn<0-7>] Somma il byte di un registro dati a un byte di memoria[1]
	ADD.W	sadr,Dn	2, 4 o 6	4(1/0)+			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] + [sadr] Somma alla word di un registro dati una word di memoria. I bit 16-31 del registro dati restano invariati[2,3]
	ADD.W	sadr,An	2, 4 o 6	8(1/0)+								[An<0-31>] ← [An<0-31>] + [sadr] (segno esteso) Somma a un registro indirizzi una word di memoria, il cui segno è esteso al formato 32 bit[2,3]
	ADD. W	Dn,Dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] + [Dn<0-15>] Somma la word di un registro dati a una word di memoria[2,3]
	ADD.L	sadr,Dn	2, 4 o 6	6(1/0)+			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] + [sadr] Somma a un registro dati una long word di memoria[2,4]
	ADD.L	sadr,An	2, 4 o 6	6(1/0)+								[An<0-31>] ← [An<0-31>] + [sadr] Somma a un registro indirizzi una long word di memoria[2,4]
	ADD.L	Dn,dadr	2, 4 o 6	14(1/2)+			X	X	X	X	X	[dadr] ← [dadr] + [Dn<0-31>] Somma un registro dati a una long word di memoria[2,4]
	ADDX.B	-(sAn),-dAn)	2	19(3/1)			X	X	X	X	X	[sAn] ← [sAn] - 1 [dAn] ← [dAn] - 1 [[dAn]] ← [[dAn]] + [[sAn]] + X Somma fra due byte di memoria con riporto (bit Extend). Entrambi gli indirizzi sono indirizzi di byte[1]
	ADDX.W	-(sAn),-(dAn)	2	19(3/1)			X	X	X	X	X	[sAn] ← [sAn] - 2 [dAn] ← [dAn] - 2 [[dAn]] ← [[dAn]] + [[sAn]] + X Somma fra due word di memoria con riporto (bit Extend). Entrambi gli indirizzi sono indirizzi di word[2,3]
	ADDX.L	-(sAn),-(dAn)	2	32(5/2)			X	X	X	X	X	[sAn] ← [sAn] - 4 [dAn] ← [dAn] - 4 [[dAn]] ← [[dAn]] + [[sAn]] + X Somma fra due long word di memoria con riporto (bit Extend). Entrambi gli indirizzi sono indirizzi di long word[2,4]
	AND.B	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	0	0	[Dn<0-7>] ← [Dn<0-7>] ∧ [sadr] AND logico di un byte di memoria al byte di un registro dati. I bit 8-31 del registro dati restano invariati[1]
	AND.B	Dn,dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [dadr] ∧ [Dn<0-7>] AND logico del byte di un registro dati a un byte di memoria[1]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Riferimenti secondari alla memoria (operazioni su memoria) (continua)	AND.W	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	0	0	[Dn<0-15>] ← [Dn<0-15>] ∧ [sadr] AND logico di una word di memoria alla word di un registro dati. I bit 16-31 del registro dati restano invariati[2,3]
	AND.W	Dn,dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [dadr] ∧ [Dn<0-15>] AND logico della word di un registro dati a una word di memoria[2,3]
	AND.L	sadr,Dn	2, 4 o 6	6(1/0)+				X	X	0	0	[Dn<0-31>] ← [Dn<0-31>] ∧ [sadr] AND logico di una long word di memoria a un registro dati[2,4]
	AND.L	Dn,dadr	2, 4 o 6	14(1/2)+				X	X	0	0	[dadr] ← [dadr] ∧ [Dn<0-31>] AND logico di un registro dati a una long word di memoria[2,4]
	CLR.B	dadr	2, 4 o 6	9(1/1)+				0	1	0	0	[dadr] ← 0 Azzera un byte di memoria[1]
	CLR.W	dadr	2, 4 o 6	9(1/1)+				0	1	0	0	[dadr] ← 0 Azzera una word di memoria[2,3]
	CLR.L	dadr	2, 4 o 6	14(1/2)+				0	1	0	0	[dadr] ← 0 Azzera una long word di memoria[2,4]
	CMP.B	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	X	X	[Dn<0-7>] − [sadr] Confronta il byte di un registro dati con un byte di memoria variando conseguentemente i codici di condizione. Registri e locazioni di memoria restano invariati in seguito a qualunque confronto[1]
	CMP.W	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	X	X	[Dn<0-15>] − [sadr] Confronta la word di un registro dati con una word di memoria variando conseguentemente i codici di condizione[2,3]
	CMP.W	sadr,An	2, 4 o 6	6(1/0)+				X	X	X	X	[An<0-15>] − [sadr] Confronta la word di un registro indirizzi con una word di memoria variando conseguentemente i codici di condizione[2,3]
	CMP.L	sadr,Dn	2, 4 o 6	6(1/0)+				X	X	X	X	[Dn<0-31>] − [sadr] Confronta un registro dati con una long word di memoria variando conseguentemente i codici di condizione[2,4]
	CMP.L	sadr,An	2, 4 o 6	6(1/0)+				X	X	X	X	[An<0-31>] − [sadr] Confronta un registroindirizzi con una long word di memoria variando conseguentemente i codici di condizione[2,4]
	CMPM.B	(sAn)+,(dAn)+	2	12(3/0)				X	X	X	X	[[dAn]] − [[sAn]] [dAn] ← [dAn] + 1 [sAn] ← [sAn] + 1 Confronta due byte di memoria variando conseguentemente i codici di condizione. I byte restano invariati[1]
	CMPM.W	(sAn)+,(dAn)+	2	12(3/0)				X	X	X	X	[[dAn]] − [[sAn]] [dAn] ← [dAn] + 2 [sAn] ← [sAn] + 2 Confronta due word di memoria variandc i codici di condizione[2,3]
	CMPM.L	(sAn)+,(dAn)+	2	20(5/0)				X	X	X	X	[[dAn]] − [[sAn]] [dAn] ← [dAn] + 4 [sAn] ← [sAn] + 4 Confronta due long word di memoria variando i codici di condizione[2,4]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

Riferimenti secondari alla memoria (operazioni su memoria) (continua)

Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
DIVS	sadr,Dn	2, 4 o 6	<158(1/0)+				X	X	X	0	[Dn<0-15>] ← [Dn<0-31>] ÷ [sadr] [Dn<16-31>] ← resto Divisione di due numeri dotati di segno. Divisione per zero causa un Trap. L'indirizzo sorgente è un indirizzo di word.[2,3]
DIVU	sadr,Dn	2, 4 o 6	≤140(1/0)+				X	X	X	0	[Dn<0-15>] ← [Dn<0-31>] ÷ [sadr] [Dn<16-31>] ← resto Divisione di due numeri non dotati di segno. Divisione per zero causa un Trap. L'indirizzo sorgente è un indirizzo di word[2,3]
EOR.B	Dn,dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [dadr] ⊻ [Dn<0-7>] OR esclusivo del byte di un registro dati a un byte di memoria[1]
EOR.W	Dn,dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [dadr] ⊻ [Dn<0-15>] OR esclusivo della word di un registro dati a una word di memoria[2,3]
EOR.L	Dn,dadr	2, 4 o 6	14(1/2)+				X	X	0	0	[dadr] ← [dadr] ⊻ [Dn<0-31>] OR esclusivo di un registro dati a una long word di memoria[2,4]
MULS	sadr,Dn	2, 4 o 6	<70(1/0)+				X	X	0	0	[Dn<0-31>] ← [Dn<0-15>] × [sadr] Moltiplicazione di due numeri a 16 bit dotati di segno, con risultato a 32 bit dotato di segno. L'indirizzo sorgente è un indirizzo di word[2,3]
MULU	sadr,Dn	2, 4 o 6	<74(2/0)+				X	X	0	0	[Dn<0-31>] ← [Dn<0-15>] × [sadr] Come sopra, per numeri non dotati di segno[2,3]
NBCD	dadr	2, 4 o 6	9(1/1)+			X	U	X	U	X	[dadr] ← 0 − [dadr] − X Negazione decimale di un byte di memoria. L'operazione fornisce il complemento a 10 se X = 0 o il complemento a 9 se X = 1.
NEG.B	dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← 0 − [dadr] Negazione di un byte di memoria[1]
NEG.W	dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← 0 − [dadr] Negazione di una word di memoria[2,3]
NEG.L	dadr	2, 4 o 6	14(1/2)+			X	X	X	X	X	[dadr] ← 0 − [dadr] Negazione di una long word di memoria[2,4]
NEGX.B	dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← 0 − [dadr] − X Negazione di un byte di memoria con bit di Extend[1]
NEGX.W	dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← 0 − [dadr] − X Negazione di una word di memoria con bit di Extend[2,3]
NEGX.L	dadr	2, 4 o 6	14(1/2)+			X	X	X	X	X	[dadr] ← 0 − [dadr] − X Negazione di una long word di memoria con bit di Extend[2,4]
NOT.B	dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [$\overline{dadr}$] Complemento a uno di un byte di memoria[1]
NOT.W	dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [$\overline{dadr}$] Complemento a uno di una word di memoria[2,3]
NOT.L	dadr	2, 4 o 6	14(1/2)+				X	X	0	0	[dadr] ← [$\overline{dadr}$] Complemento a uno di una long word di memoria[2,4]
OR.B	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	0	0	[Dn<0-7>] ← [Dn<0-7>] ∨ [sadr] OR logico di un byte di memoria al byte di un registro dati. I bit 8-31 del registro dati restano invariati[1]
OR.B	Dn,dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [dadr] ∨ [Dn<0-7>] OR logico del byte di un registro dati a un byte di memoria[1]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Riferimenti secondari alla memoria (operazioni su memoria) (continua)	OR.W	sadr,Dn	2, 4 o 6	4(1/0)+				X	X	0	0	[Dn<0-15>] ← [Dn<0-15>] ∨ [sadr] OR logico di una word di memoria alla word di un registro dati. I bit 16-31 del registro dati restano invariati[2,3]
	OR.W	Dn,dadr	2, 4 o 6	9(1/1)+				X	X	0	0	[dadr] ← [dadr] ∧ [Dn<0-15>] OR logico della word di un registro dati a una word di memoria[2,3]
	OR.L	sadr,Dn	2, 4 o 6	6(1/0)+				X	X	0	0	[Dn<0-31>] ← [Dn<0-31>] ∨ [sadr] OR logico di una long word di memoria a un registro dati[2,4]
	OR.L	Dn,dadr	2, 4 o 6	14(1/2)+				X	X	0	0	[dadr] ← [dadr] ∧ [Dn<0-31>] OR logico di un registro dati a una long word di memoria[2,4]
	SBCD	-(sAn),-(dAn)	2	19(3/1)			X	U	X	U	X	[sAn] ← [sAn] - 1 [dAn] ← [dAn] - 1 [[dAn]] ← [[dAn]] - [[sAn]] - X Sottrazione decimale fra due byte di memoria con riporto (bit Extend). Entrambi gli indirizzi sono indirizzi di byte[1]
	SCC	dadr	2, 4 o 6	9(1/1)+								[dadr] ← tutti 1 se cc = vero [dadr] ← tutto 0 se cc = falso Pone un byte di memoria uguale allo stato[1]
	SUB.B	sadr,Dn	2, 4 o 6	4(1/0)+			X	X	X	X	X	[Dn<0-7>] ← [Dn<0-7>] - [sadr] Sottrae un byte di memoria dal byte di un registro dati. I bit 8-31 del registro dati restano invariati[1]
	SUB.B	Dn,dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] - [Dn<0-7>] Sottrae il byte di un registro dati da un byte di memoria[1]
	SUB.W	sadr,Dn	2, 4 o 6	4(1/0)+			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] - [sadr] Sottrae una word di memoria dalla word di un registro dati. I bit 16-31 del registro dati restano invariati[2,3]
	SUB.W	sadr,An	2, 4 o 6	8(1/0)+			X	X	X	X	X	[An<0-31>] ← [An<0-31>] - [sadr] (segno esteso) Sottrae una word di memoria da un registro indirizzi. Il segno della word di memoria è esteso a 32 bit[2,3]
	SUB.W	Dn,dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] - [Dn<015>] Sottrae la word da un registro dati da una word di memoria[2,1]
	SUB.L	sadr,Dn	2, 4 o 6	6(1/0)+			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] - [sadr] Sottrae una long word di memoria da un registro dati[2,4]
	SUB.L	sadr,An	2, 4 o 6	6(1/0)+			X	X	X	X	X	[An<0-31>] ← [An<0-31>] - [sadr] Sottrae una long word di memoria da un registro indirizzi[2,4]
	SUB.L	Dn,dadr	2, 4 o 6	14(1/2)+			X	X	X	X	X	[dadr] ← [dadr] - [Dn<0-31>] Sottrae un registro dati da una long word di memoria[2,4]
	SUBX.B	-(sAn),-(dAn)	2	19(3/1)			X	X	X	X	X	[sAn] ← [sAn] - 1 [dAn] ← [dAn] - 1 [[dAn]] ← [[dAn]] - [[sAn]] - X Sottrazione fra due byte di memoria con prestito (bit Extend). Entrambi gli indirizzi sono indirizzi di byte[1]
	SUBX.W	-(sAn),-(dAn)	2	19(3/1)			X	X	X	X	X	[sAn] ← [sAn] - 2 [dAn] ← [dAn] - 2 [[dAn]] ← [[dAn]] - [[sAn]] - X Sottrazione fra due word di memoria con prestito (bit Extend). Entrambi gli indirizzi sono indirizzi di word[2,3]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Riferim. secondari alla memoria (operaz. su memoria) (continua)	SUBX.L	-(sAn),-(dAn)	2	32(5/2)			X	X	X	X	X	[sAn] ← [sAn] - 4 [dAn] ← [dAn] - 4 [[dAn]] ← [[dAn]] - [[sAn]] - X Sottrazione fra due long word di memoria con prestito (bit Extend). Entrambi gli indirizzi sono indirizzi di long word[2,4]
	TAS	dadr	2, 4 o 6	11(1/1)+				X	X	0	0	[dadr<7>] ← 1 Testa lo stato di un byte di memoria e pone a 1 il bit più significativo
	TST.B	dadr	2, 4 o 6	4(1/0)+				X	X	0	0	[dadr] - 0 Testa lo stato di un byte di memoria lasciandolo invariato
	TST.W	dadr	2, 4 o 6	4(1/0)+				X	X	0	0	[dadr] - 0 Testa lo stato di una word di memoria lasciandola invariata
	TST.L	dadr	2, 4 o 6	4(1/0)+				X	X	0	0	[dadr] - 0 Testa lo stato di una long word di memoria lasciandola invariata
Operando immediato[5]	MOVEQ	data8,Dn	2	4(1/0)				X	X	0	0	[Dn<0-7>] ← data8 [Dn<8-32>] ← [Dn<7>] Pone in un registro dati un byte di dati immediati. Il segno è esteso a tutti i bit più significativi del registro
	MOVE.B	data8,Dn	4	8(2/0)				X	X	0	0	[Dn<0-7>] ← data8 Pone in un registro dati un byte di dati immediati. I bit 8-31 del registro restano invariati
	MOVE.B	data8,dadr	4, 6 o 8	9(1/1)+				X	X	0	0	[dadr] ← [data8] Pone in una locazione di memoria un byte di dati immediati[1]
	MOVE.W	data16,Dn	4	8(2/0)				X	X	0	0	[Dn<0-15>] ← data16 Pone in un registro dati una word di dati immediati. I bit 16-31 del registro dati restano invariati
	MOVE.W	data16,An	4	8(2/0)								[An<0-15>] ← data16 [An<16-31>] ← [An<15>] Pone in un registro indirizzi una word di dati immediati. Il segno è esteso a tutti i bit più significativi del registro[5]
	MOVE.W	data16,dadr	4, 6 o 8	9(1/1)+				X	X	0	0	[dadr] ← data16 Pone una word di dati immediati in una locazione di memoria[2,3]
	MOVE.L	data32,Dn	6	12(3/0)				X	X	0	0	[Dn<0-31>] ← data32 Pone una long word di dati immediati in un registro dati
	MOVE.L	data32,An	6	12(3/0)								[An<0-31>] ← data32 Pone una long word di dati immediati in un registro indirizzi
	MOVE.L	data32,dadr	6, 8 o 10	18(2/2)+				X	X	0	0	[dadr] ← data32 Pone una long word di dati immediati in una locazione di memoria[2,4]
Operazioni immediate[5]	ADD.B	data8,Dn	4	8(2/0)			X	X	X	X	X	[Dn<0-7>] ← [Dn<0-7>] + data8 Somma un byte di dati immediati al byte di un registro dati. I bit 8-31 del registro dati restano invariati
	ADD.B	data8,dadr	4, 6 o 8	13(2/1)+			X	X	X	X	X	[dadr] ← [dadr] + data8 Somma un byte di dati immediati a un byte di memoria[1]
	ADD.W	data16,Dn	4	8(2/0)			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] + data16 Somma una word di dati immediati alla word di un registro dati. I bit 16-31 del registro dati restano invariati

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Operazioni immediate[5] (continua)	ADD.W	data16,An	4	8(2/0)								[An<0-31>] ← [An<0-31>] + data 16 (segno esteso) Somma una word di dati immediati a un registro indirizzi. Il segno della word di dati è esteso, per l'operazione, a 32 bit
	ADD.W	data16,dadr	4, 6 o 8	13(2/1)+			X	X	X	X	X	[dadr] ← [dadr] + data16 Somma una word di dati immediati a una locazione di memoria[2,3]
	ADD.L	data32,Dn	6	16(3/0)			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] + data32 Somma una long word di dati immediati a un registro dati
	ADD.L	data32,An	6	16(3/0)								[An<0-31>] ← [An<0-31>] + data32 Somma una long word di dati immediati a un registro indirizzi
	ADD.L	data32,dadr	6, 8 o 10	22(3/2)+			X	X	X	X	X	[dadr] ← [dadr] + data32 Somma long word di dati immediati a long word di memoria[2,4]
	ADDQ.B	data3,Dn	2	4(1/0)			X	X	X	X	X	[Dn<0-7>] ← [Dn<0-7>] + data3 Somma tre bit immediati al byte di un registro dati. I bit 8-31 del registro restano invariati
	ADDQ.B	data3,dadr	2, 4 o 6	9(1/0)+			X	X	X	X	X	[dadr] ← [dadr] + data3 Somma tre bit immediati a un byte di memoria[1]
	ADDQ.W	data3,Dn	2	4(1/0)			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] + data3 Somma tre bit immediati alla word di un registro dati. I bit 16-31 del registro restano invariati
	ADDQ.W	data3,An	2	4(1/0)								[An<0-15>] ← [An<0-15>] + data3 Somma tre bit immediati alla word di un registro indirizzi. I bit 16-31 del registro restano invariati
	ADDQ.W	data3,dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] + data3 Somma tre bit immediati a una word di memoria[2,3]
	ADDQ.L	data3,Dn	2	8(1/0)			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] + data3 Somma tre bit immediati a un registro dati
	ADDQ.L	data3,An	2	8(1/0)								[An<0-31>] ← [An<0-31>] + data3 Somma tre bit immediati a un registro indirizzi
	ADDQ.L	data3,dadr	2, 4 o 6	14(1/2)			X	X	X	X	X	[dadr] ← [dadr] + data3 Somma tre bit immediati a una long word di memoria[2,4]
	AND.B	data8,Dn	4	8(2/0)				X	X	0	0	[Dn<0-7>] ← [Dn<0-7>] ∧ data8 AND logico di un byte di dati immediati al byte di un registro dati. I bit 8-31 del registro restano invariati
	AND.B	data8,dadr	4, 6 o 8	13(2/1)+				X	X	0	0	[dadr] ← [dadr] ∧ data8 AND logico di un byte di dati immediati a un byte di memoria[1]
	AND.W	data16,Dn	4	8(2/0)				X	X	0	0	[Dn<0-15>] ← [Dn<0-15>] ∧ data16 AND logico di una word di dati immediati alla word meno significativa di un registro dati. I bit 16-31 del registro restano invariati
	AND.W	data16,dadr	4, 6 o 8	13(2/1)				X	X	0	0	[dadr] ← [dadr] ∧ data16 AND logico di una word di dati immediati a una word di memoria[2,3]
	AND.L	data32,Dn	6	16(3/0)				X	X	0	0	[Dn<0-31>] ← [Dn<0-31>] ∧ data32 AND logico di una long word di dati immediati a un registro dati
	AND.L	data32,dadr	6, 8 o 10	22(3/2)+				X	X	0	0	[dadr] ← [dadr] ∧ data32 AND logico di una word di dati immediati a una long word di memoria[2,4]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato							Operazione eseguita
					T	S	X	N	Z	V	C	
Operazioni immediate[5] (continua)	CMP.B	data8,Dn	4	8(2/0)				X	X	X	X	[Dn<0-7>] - data8 Confronta il byte di un registro dati con un byte di dati immediati variando conseguentemente i codici di condizione. I dati presenti nel registro restano invariati
	CMP.B	data8,dadr	4, 6 o 8	8(2/0)+				X	X	X	X	[dadr] - data8 Confronta un byte di memoria con un byte di dati immediati variando conseguentemente i codici di condizione[1]
	CMP.W	data16,Dn	4	8(2/0)				X	X	X	X	[Dn<0-15>] - data16 Confronta la word di un registro dati con una word di dati immediati variando conseguentemente i codici di condizione
	CMP.W	data16,An	4	8(2/0)				X	X	X	X	[An<0-15>] - data 16 Confronta la word di un registro indirizzi con una word di dati immediati variando conseguentemente i codici di condizione
	CMP.W	data16,dadr	4, 6 o 8	8(2/0)+				X	X	X	X	[dadr] - data16 Confronta una word di memoria con una word di dati immediati variando conseguentemente i codici di condizione[2,3]
	CMP.L	data32,Dn	6	14(3/0)				X	X	X	X	[Dn<0-31>] - data32 Confronta un registro dati con una long word di dati immediati variando conseguentemente i codici di condizione
	CMP.L	data32,An	6	14(3/0)				X	X	X	X	[An<0-31>] - data32 Confronta un registro indirizzi con una long word di dati immediati variando conseguentemente i codici di condizione
	CMP.L	data32,dadr	6, 8 o 10	12(3/0)+				X	X	X	X	[dadr] - data32 Confronta una long word di memoria con una long word di dati immediati variando conseguentemente i codici di condizione[2,4]
	DIVS	data16,Dn	4	≤162(2/0)				X	X	X	0	[Dn<0-15>] ← [Dn<0-31>] ÷ data16 [Dn<16-31>] ← resto Divisione fra numeri dotati di segno. Divis. per zero causa un Trap
	DIVU	data16,Dn	4	≤148(2/0)				X	X	X	0	[Dn<0-15>] ← [Dn<0-31>] ÷ data16 [Dn<16-31>] ← resto Divis. fra num. non dotati di segno. Disegno per zero causa un Trap
	EOR.B	data8,Dn	4	8(2/0)				X	X	0	0	[Dn<0-7>] ← [Dn<0-7>] ⊻ data8 OR esclusivo di un byte di dati immediati a un registro dati. I bit 8-31 del registro restano invariati
	EOR.B	data8,dadr	4, 6 o 8	13(2/1)+				X	X	0	0	[dadr] ← [dadr] ⊻ data8 OR esclusivo di un byte di dati immediati a un byte di memoria[1]
	EOR.W	data16,Dn	4	8(2/0)				X	X	0	0	[Dn<0-15>] ← [Dn<0-15>] ⊻ data16 OR esclusivo di una word di dati immediati alla word di un registro dati. I bit 16-31 del registro restano invariati
	EOR.W	data16,dadr	4, 6 o 8	13(2/1)+				X	X	0	0	[dadr] ← [dadr] ⊻ data16 OR esclusivo di una word di dati immediati a una word di memoria[2,3]
	EOR.L	data32,Dn	6	16(3/0)				X	X	0	0	[Dn<0-31>] ← [Dn>0-31>] ⊻ data32 OR esclusivo di una long word di dati immediati a un registro dati
	EOR.L	data32,dadr	6, 8 o 10	22(3/2)+				X	X	0	0	[dadr] ← [dadr] ⊻ data32 OR esclusivo di una long word di dati immediati a una long word di memoria[2,4]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	Stato S	Stato X	Stato N	Stato Z	Stato V	Stato C	Operazione eseguita
Operazioni immediate[5] (continua)	MULS	data16,Dn	4	≤74(2/0)				X	X	O	O	[Dn<0-311>] ← [Dn<0-15>] x data16 Moltiplicazione di due numeri a 16 bit con segno; risultato a 32 bit con segno
	MULU	data16,Dn	4	≤74(2/0)				X	X	O	O	[Dn<0-31>] ← [Dn<0-15>] x data16 Moltiplicazione di due numeri a 16 bit non dotati di segno, con risultato a 32 bit non dotato di segno
	OR.B	data8,Dn	4	8(2/0)				X	X	O	O	[Dn<0-7>] ← [Dn<0-7>] V data8 OR logico di un byte di dati immediati al byte di un registro dati. I bit 8-31 del registro restano invariati
	OR.B	data8,dadr	4, 6 o 8	13(2/1)+				X	X	O	O	[dadr] ← [dadr] V data8 OR logico di un byte di dati immediati a un byte di memoria[1]
	OR.W	data16,Dn	4	8(2/0)				X	X	O	O	[Dn<0-15>] ← [Dn<0-15>] V data16 OR logico di una word di dati immediati alla word di un registro dati. I bit 16-31 del registro restano invariati
	OR.W	data16,dadr	4, 6 o 8	13(2/1)+				X	X	O	O	[dadr] ← [dadr] V data16 OR logico di una word di dati immediati a una word di memoria[2,3]
	OR.L	data32,Dn	6	16(3/0)				X	X	O	O	[Dn<0-31>] ← [Dn<0-31>] V data32 OR logico di una long word di dati immediati a un registro dati
	OR.L	data32,dadr	6, 8 o 10	22(3/2)+				X	X	O	O	[dadr] ← [dadr] V data32 OR logico di long word di dati immediati a long word di memoria[2,4]
	SUB.B	data8,Dn	4	8(2/0)			X	X	X	X	X	[Dn<0-7>] ← [Dn<0-7>] - data8 Sottrazione di un byte di dati immediati da un registro dati. I bit 8-31 del registro restano invariati
	SUB.B	data8,dadr	4, 6 o 8	13(2/1)+			X	X	X	X	X	[dadr] ← [dadr] - data8 Sottrazione di un byte di dati immediati da un byte di memoria[1]
	SUB.W	data16,Dn	4	8(2/0)			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] - data16 Sottrazione di una word di dati immediati dalla word di un registro dati. I bit 16-31 del registro restano invariati
	SUB.W	data16,An	4	8(2/0)								[An <0-31>] ← [An<0-31>] - data16 segno esteso Sottrazione di una word di dati immediati da un registro indirizzi. Il segno della word di dati è esteso, per l'operazione, a 32 bit
	SUB.W	data16,dadr	4, 6 o 8	13(2/1)+			X	X	X	X	X	[dadr] ← [dadr] - data16 Sottrazione di una word di dati immediati da una word di memoria[2,3]
	SUB.L	data32,Dn	6	16(3/0)			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] - data32 Sottrazione di una long word di dati immediati da un registro dati
	SUB.L	data32,An	6	16(3/0)								[An<0-31>] ← [An<0-31>] - data32 Sottrazione di long word di dati immediati da un registro indirizzi
	SUB.L	data32,dadr	6, 8 o 10	22(3/2)+			X	X	X	X	X	[dadr] ← [dadr] - data32 Sottrazione di long word di dati immediati da long word di memoria[2,4]
	SUBQ.B	data3,Dn	2	4(1/0)			X	X	X	X	X	[Dn<0-7>] ← [Dn<0-7>] - data3 Sottrazione di tre bit immediati dal byte di un registro dati. I bit 8-31 del registro restano invariati
	SUBQ.B	data3,dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] - data3 Sottrazione di tre bit immediati da un byte di memoria[1]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	Stato S	Stato X	Stato N	Stato Z	Stato V	Stato C	Operazione eseguita
Operazioni immediate[5] (continua)	SUBQ.W	data3,Dn	2	4(1/0)			X	X	X	X	X	[Dn<0-15>] ← [Dn<0>] - data3 Sottrazione di tre bit immediati dalla word di un registro dati. I bit 16-31 del registro restano invariati
	SUBQ.W	data3,An	2	4(1/0)								[An<0-15>] ← [An<0-15>] - data3 Sottrazione di tre bit immediati dalla word di un registro indirizzi. I bit 16-31 del registro restano invariati
	SUBQ.W	data3,dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	[dadr] ← [dadr] - data3 Sottrazione di tre bit immediati da una word di memoria[2,3]
	SUBQ.L	data3,Dn	2	8(1/0)			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] - data3 Sottrazione di tre bit immediati da un registro dati
	SUBQ.L	data3,An	2	8(1/0)								[An<0-31>] ← [An<0-31>] - data3 Sottrazione di tre bit immediati da un registro indirizzi
	SUBQ.L	data3,dadr	2, 4 o 6	14(1/2)+			X	X	X	X	X	[dadr] ← [dadr] - data3 Sottrazione di tre bit immediati da una long word di memoria[2,4]
JUMP BRANCH	BRA	label	2 o 4	10(2/0)								[PC] ← label Diramazione incondizionata (breve)
	JMP	jadr	2, 4 o 6	4(1/0)+								[PC] ← jadr Salto incondizionato
Chiamata a subroutine e ritorno	BSR	label	2 o 4	10, 8(1/0) 10, 12(2/0)								[A7] ← [A7] - 2 [[A7]] ← [PC] [PC] ← label Diramazione a subroutine (breve)
	JSR	jadr	2, 4 o 6	14(1/2)+								[A7] ← [A7] - 2 [[A7]] ← [PC] [PC] ← jadr Salto a subroutine
	RTS		2	16(4/0)								[PC] ← [[A7]] [A7] ← [A7] + 2 Ritorno da subroutine
	RTR		2	20(5/0)								[SR<0-4>] ← [[A7<0-4>]] [A7] ← [A7] + 2 [PC] ← [[A7]] [A7] ← [A7] + 2 Ripristino dei codici di condizione e ritorno da subroutine
Diramazione condizionale	Bcc	label	2 o 4	10, 8(1/0) 10, 12(2/0)								[PC] ← label Diramazione condizionale
	DBcc	Dn,label	4	12(2/0) 10(2/0), 14(3/0)								Se cc, nessuna azione successiva [Dn<0-15>] ← [Dn<0-15>] - 1 If [Dn<0-15>] = nessuna azione successiva [PC] ← label Testa la condizione, decrementa ed esegue la diramazione. Continua in ciclo fino al verificarsi della condizione, o fino all'esaurimento del contatore di loop.

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Movimento dati fra registri	MOVE.B	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<07>] ← [sDn<0-7>] Copia il byte di un registro dati nel byte di un altro registro dati. I bit 8-31 del registro "destinazione" restano invariati
	MOVE.W	rs,Dn	2	4(1/0)				X	X	0	0	[Dn<0-15>] ← [rs<0-15>] Copia la word di un registro dati nella word di un altro registro dati. I bit 16-31 del registro "destinazione" restano invariati
	MOVE.W	rs,An	2	4(1/0)								[An<015>] ← [rs<0-15>] [An<16-31>] ← [An<15>] Copia la word di un registro dati o indirizzi nella word di un registro indirizzi. Il segno è esteso ai bit più significativi del registro indirizzi
	MOVE.L	rs,Dn	2	4(1/0)				X	X	0	0	[Dn<0-31>] ← [rs<0-31>] Copia un registro dati o indirizzi in un registro dati
	MOVE.L	rs,An	2	4(1/0)								[An<0-31>] ← [rs<0-31>] Copia un registro dati o indirizzi in un registro indirizzi
Operazioni fra registri	ABCD	sDn,dDn	2	6(1/0)			X	U	X	U	X	[dDn<0-7>] ← [dDn<0-7>] + [sDn<0-7>] + X Somma decimale del byte del registro dati "sorgente" al byte del registro dati "destinazione" con riporto (bit Extend). I bit 8-31 del registro "destinazione" restano invariati
	ADD.B	sDn,dDn	2	4(1/0)			X	X	X	X	X	[dDn<0-7>] ← [dDn<0-7>] + [sDn<0-7>] Somma del byte del reg. dati "sorgente" al byte del reg. dati "destinazione". I bit 8-31 del registro "destinazione" restano invariati
	ADD.W	rs,Dn	2	4(1/0)			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] + [rs<0-15>] Somma della word del reg. "sorgente" alla word del reg. dati "destinazione". I bit 16-31 del registro "destinazione" restano invariati
	ADD.W	rs,An	2	8(1/0)								[An<0-15>] ← [An<0-15>] + [rs<0-15>] (segno esteso) Somma della word del reg. "sorgente" alla word del reg. indirizzi "destinazione". Il segno della word "sorgente" è esteso a 32 bit
	ADD.L	rs,Dn	2	8(1/0)			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] + [rs<0-31>] Somma di un registro "sorgente" a un registro dati
	ADD.L	rs,An	2	8(1/0)								[An<0-31>] ← [An<0-31>] + rs<0-31>] Somma di un registro "sorgente" a un registro indirizzi
	ADDX.B	sDn,dDn	2	4(1/0)			X	X	X	X	X	[dDn<0-7>] ← [dDn<0-7] + [sDn<0-7>] + X Somma il byte del registro dati "sorgente" al byte del registro dati "destinazione" con riporto (bit Extend). I bit 8-31 del registro dati "destinazione" restano invariati
	ADDX.W	sDn, dDn	2	4(1/0)			X	X	X	X	X	[dDn<0-15>] ← [dDn<0-15>] + [sDn<0-15>] + X Somma della word del registro dati "sorgente" alla word del registro dati "destinazione" con riporto (bit Extend). I bit 16-31 del registro "destinazione" restano invariati
	ADDX.L	sDn,dDn	2	8(1/0)			X	X	X	X	X	[dDn<0-31>] ← [dDn<0-31>] + [sDn<0-31>] + X Somma il registro dati "sorgente" al registro dati "destinazione" con riporto (bit Extend)
	AND.B	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<0-7>] ← [dDn<0-7>] ∧ [sDn<0-7>] AND logico del byte del reg. dati "sorgente" al byte del reg. dati "destinazione". I bit 8-31 del reg. dati "destinazione" restano invariati

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	• Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Operazioni fra registri (continua)	AND.W	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<0-15>] ← [dDn<0-15>] ∧ [sDn<0-15>] AND logico della word del reg. dati "sorgente" alla word del reg. dati "destinazione". I bit 16-31 del registro "destinazione" restano invariati
	AND.L	sDn,dDn	2	8(1/0)				X	X	0	0	[dDn<0-31>] ← [dDn<0-31>] ∧ [sDn<0-31>] AND logico del registro dati "sorgente" al registro dati "destinazione"
	CMP.B	sDn,dDn	2	4(1/0)				X	X	X	X	[dDn<0-7>] − [sDn<0-7>] Confronta il byte del registro dati "sorgente" col byte del registro dati "destinazione" variando i codici di condizione. I dati restano invariati
	CMP.W	rs,Dn	2	4(1/0)				X	X	X	X	[Dn<0-15>] − [rs<0-15>] Confronta la word del registro dati "destinazione" con la word di un registro "sorgente", variando i codici di condizione
	CMP.W	rs,An	2	6(1/0)				X	X	X	X	[An<0-15>] − [rs<0-15>] Confronta la word del registro indirizzi "destinazione" con la word di un registro "sorgente", variando i codici di condizione
	CMP.L	rs,Dn	2	6(1/0)				X	X	X	X	[Dn<0-31>] − [rs<0-31>] Confronta il registro dati "destinazione" con registro "sorgente", variando i codici di condizione
	CMP.L	rs,An	2	6(1/0)				X	X	X	X	[An<0-31>] − [rs<0-31>] Confronta il registro indirizzi "destinazione" col registro "sorgente", variando i codici di condizione
	DIVS	sDn,dDn	2	≤158(1/0)				X	X	X	0	[dDn<0-15>] ← [dDn<0-31>] ÷ [sDn<0-15>] [dDn<016-31>] ← resto Divisione fra numeri dotati di segno. Divisione per zero causa un Trap
	DIVU	sDn,dDn	2	≤140(1/0)				X	X	X	0	[dDn<0-15>] ← [dDn<0-31>] + [sDn<0-15>] [dDn<16-31>] ← resto Divis. fra numeri dotati di segno. Divis. per zero causa un Trap
	EOR.B	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<0-7>] ← [dDn<0-7>] ⊻ [sDn<0-7>] OR esclusivo del byte del reg. dati "sorgente" al byte del reg. dati "destinazione". I bit 8-31 del registro "destinazione" restano invariati
	EOR.W	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<0-15>] ← [dDn<0-15>] ⊻ [sDn<0-15>] OR esclusivo del byte del reg. dati "sorgente" al byte del reg. dati "destinazione". I bit 16-31 del reg. "destinazione" restano invariati
	EOR.L	sDn,dDn	2	8(1/0)				X	X	0	0	[dDn<0-31>] ← [dDn<0-31>] ⊻ [sDn<0-31>] OR esclusivo del reg. dati "sorgente" al reg. dati "destinazione"
	EXG	rs,rd	2	6(1/0)								[rd] ←→ [rs] Scambio del contenuto di due registri. Questa istruzione opera sempre in formato long word (32 bit)
	MULS	sDn,dDn	2	≤70(1/0)				X	X	0	0	[dDn<0-31>] ← [dDn<0-15>] × [sDn<0-15>] Moltiplicaz. fra numeri a 16 bit con segno; risultato a 32 bit con segno
	MULU	sDn,dDn	2	≤70(1/0)				X	X	0	0	[dDn<0-31>] ← [dDn<0-15>] × [sDn<0-15>] Moltiplicazione fra numeri a 16 bit non dotati di segno; risultato a 32 bit non dotato di segno
	OR.B	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<0-7>] ← [dDn<0-7>] ∨ [sDn<0-7>] OR logico del byte del registro dati "sorgente" al byte del registro dati "destinazione". I bit 8-31 del registro dati "destinazione" restano invariati

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Operazioni fra registri (continua)	OR.W	sDn,dDn	2	4(1/0)				X	X	0	0	[dDn<0-15>] ← [dDn<0-15>] V [sDn<0-15>] OR logico della word del reg. dati "sorgente" alla word del reg. dati "destinazione". I bit 16-31 del registro "destinazione" restano invariati
	OR.L	sDn,dDn	2	8(1/0)				X	X	0	0	[dDn<0-31>] ← [dDn<0-31>] V [sDn<0-31>] OR logico del registro dati "sorgente" al registro dati "destinazione"
	SBCD	sDn,dDn	2	6(1/0)			X	U	X	U	X	[dDn<0-7>] ← [dDn<0-7>] − [sDn<0-7>] − X Sottrazione decimale del byte del registro dati "sorgente" dal byte del registro dati "destinazione" con riporto (bit Extend). I bit 8-31 del registro "destinazione" restano invariati
	SUB.B	sDn,dDn	2	4(1/0)			X	X	X	X	X	[dDn<0-7>] ← [dDn<0-7>] − [sDn<0-7>] Sottrazione del byte del reg. dati "sorgente" dal byte del reg. dati "destinazione". I bit 8-31 del registro "destinazione" restano invariati
	SUB.W	rs,Dn	2	4(1/0)			X	X	X	X	X	[Dn<0-15>] ← [Dn<0-15>] − [rs<0-15>] Sottrazione della word del reg. "sorgente" dalla word del reg. dati "destinazione". I bit 16-31 del reg. "destinazione" restano invariati
	SUB.W	rs,An	2	8(1/0)								[An<0-15>] ← [An<0-15>] − [rs<0-15>] (segno esteso) Sottrazione della word del reg. "sorgente" dalla word di un reg. indirizzi. Il segno della word "sorgente" viene esteso a 32 bit
	SUB.L	rs,Dn	2	8(1/0)			X	X	X	X	X	[Dn<0-31>] ← [Dn<0-31>] − [rs<0-31>] Sottrazione del reg. "sorgente" dal reg. dati "destinazione"
	SUB.L	rs,An	2	8(1/0)								[An<0-31>] ← [An<0-31>] − [rs<0-31>] Sottrazione del reg. "sorgente" dal reg. indirizzi "destinazione"
	SUBX.B	sDn,dDn	2	4(1/0)			X	X	X	X	X	[dDn<0-7>] ← [dDn<0-7>] − [sDn<0-7>] − X Sottrazione del byte del registro dati "sorgente" dal byte del registro dati "destinazione" con prestito (bit Extend). I bit 8-31 del registro dati destinazione restano invariati
	SUBX.W	sDn,dDn	2	4(1/0)			X	X	X	X	X	[dDn<0-15>] ← [dDn<0-15>] − [sDn<0-15>] − X Sottrazione della word del registro dati "sorgente" dalla word del registro dati "destinazione" con prestito (bit Extend). I bit 16-31 del registro dati "destinazione" restano invariati
	SUBX.L	sDn,dDn	2	8(1/0)			X	X	X	X	X	[dDn<0-31>] ← [dDn<0-31>] − [sDn<0-31>] − X Sottrazione del registro dati "sorgente" dal registro dati "destinazione" con prestito (bit Extend)
Operazioni su registri	CLR.B	Dn	2	4(1/0)				0	1	0	0	[Dn<0-7>] ← 0 Azzera il byte di un registro dati. I bit 8-31 del registro restano invariati
	CLR.W	Dn	2	4(1/0)				0	1	0	0	[Dn<0-15> ← 0 Azzera la word di un registro dati. I bit 16-31 del registro restano invariati
	CLR.L	Dn	2	6(1/0)				0	1	0	0	[Dn<0-31>] ← 0 Azzera un registro dati
	EXT.W	Dn	2	4(1/0)				X	X	0	0	[Dn<8-15>] ← [Dn<7>] Estende il bit di segno di un byte di dati alla misura word. I bit 16-31 del registro dati restano invariati
	EXT.L	Dn	2	4(1/0)				X	X	0	0	[Dn<16-31>] ← [Dn<15>] Estende il bit di segno di una word di dati alla misura long word

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Operazioni fra registri (continua)	NBCD	Dn	2	6(1/0)			X	U	X	U	X	[Dn <0-7>] ← [Dn <0-7>] – X Negazione decimale del byte di un registro dati. I bit 8-31 del registro dati restano invariati
	NEG.B	Dn	2	4(1/0)			X	X	X	X	X	[Dn <0>] ← 0 – [Dn <0-7>] Negazione del byte di un reg. dati. I bit 8-31 del reg. restano invariati
	NEG.W	Dn	2	4(1/0)			X	X	X	X	X	[Dn <0-15>] ← 0 – [Dn <0-15>] Negazione della word di un reg. dati. I bit 16-31 del reg. restano invariati
	NEG.L	Dn	2	6(1/0)			X	X	X	X	X	[Dn <0-31>] ← 0 – [Dn <0-31>] Negazione di un registro
	NEG.B	Dn	2	4(1/0)			X	X	X	X	X	[Dn <0-7>] ← 0 – [Dn <0-7] – X Negazione del byte di un registro dati con Extend. I bit 8-31 del registro restano invariati
	NEG.W	Dn	2	4(1/0)			X	X	X	X	X	[Dn <0-15>] ← 0 – [Dn <0-15>] – X Negazione della word di un registro dati con Extend. I bit 16-31 del registro dati restano invariati
	NEG.L	Dn	2	6(1/0)			X	X	X	X	X	[Dn <0-31>] ← 0 – [Dn <0-31>] – X Negazione di un registro con Extend
	NOT.B	Dn	2	4(1/0)				X	X	0	0	[Dn <0-7>] ← $\overline{\text{[Dn<0-7>]}}$ Complemento a uno del byte di un registro dati. I bit 8-31 del registro dati restano invariati
	NOT.W	Dn	2	6(1/0)				X	X	0	0	[Dn <0-15>] ← $\overline{\text{[Dn<0-15>]}}$ Complemento a uno della word di un registro dati. I bit 16-31 del registro dati restano invariati
	NOT.L	Dn	2	6(1/0)				X	X	0	0	[Dn <0-3>] ← $\overline{\text{[Dn<0-31>]}}$ Complemento a uno di un registro dati
	Scc	Dn	2	9(1/1)								[Dn <0-7>] ← tutti 1 se cc = VERO [Dn <0-1>] ← tutti 0 se cc = FALSO Pone lo stato nel byte di un registro dati
	SWAP	Dn	2	4(1/0)				X	X	0	0	[Dn <0-15>] ←→ [Dn <16-31>] Scambia i contenuti delle due word di un registro dati
	TAS	Dn	2	4(1/0)				X	X	0	0	[Dn <7>] ← 1 Testa lo stato del byte di un registro dati e pone a uno il bit 7
	TST.B	Dn	2	4(1/0)				X	X	0	0	[Dn <0-7>] – 0 Testa lo stato del byte di un registro dati, il cui contenuto resta invariato
	TST.W	Dn	2	4(1/0)				X	X	0	0	[Dn <0-15>] – 0 Testa lo stato della word di un registro dati, il cui contenuto resta invariato
	TST.L	Dn	2	4(1/0)				X	X	0	0	[Dn <0-31>] – 0 Testa lo stato di un registro dati, il cui contenuto resta invariato

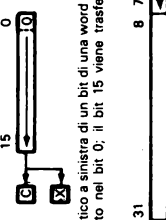
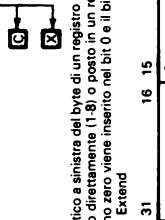
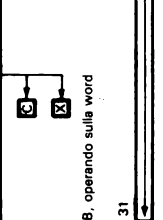
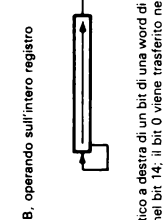
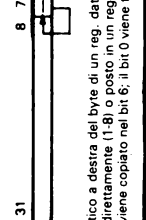
Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Shift	ASL	dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	Shift aritmetico a sinistra di un bit di una word di memoria. Uno zero viene inserito nel bit 0; il bit 15 viene trasferito nei bit Carry ed Extend[2,3]
	ASL.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	X X	X X	Shift aritmetico a sinistra del byte di un registro dati. Il numero di shift è specificato direttamente (1-8) o posto in un registro dati (1-63). Ad ogni shift uno zero viene inserito nel bit 0 e il bit 7 viene trasferito nei bit Carry ed Extend
	ASL.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	X X	X X	Come ASL.B, operando sulla word
	ASL.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)			X X	X X	X X	X X	X X	Come ASL.B, operando sull'intero registro
	ASR	dadr	2, 4 o 6	9(1/1)+			X	X	X	X	X	Shift aritmetico a destra di un bit di una word di memoria. Il bit 15 viene copiato nel bit 14; il bit 0 viene trasferito nei bit Carry ed Extend
	ASR.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	X X	X X	Shift aritmetico a destra del byte di un reg. dati. Il numero di shift è specificato direttamente (1-8) o posto in un reg. dati (1-63). Ad ogni shift il bit 7 viene copiato nel bit 6; il bit 0 viene trasferito nei bit Carry ed Extend

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Shift (continua)	ASR.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	X X	X X	31 16 15 0 C X Come ASR.B, operando sulla word
	ASR.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)			X X	X X	X X	X X	X X	31 0 C X Come ASR.B, operando sull'intero registro
	LSL	dadr	2, 4 o 6	9(1/1)+			X	X	X	0	X	15 0 C 0 X Shift logico a sinistra di un bit di una word di memoria. Uno zero viene inserito nel bit 0; il bit 15 viene trasferito nei bit Carry ed Extend. (LSL è identico a ASL tranne che per lo stato del bit Overflow)[2, 3]
	LSL.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	0 0	X X	31 8 7 0 0 C X Shift logico a sinistra del byte meno significativo di un registro dati. Il numero di shift è specificato direttamente (1-8) o posto in un registro dati (1-63). Ad ogni shift uno zero viene inserito nel bit 0 e il bit 7 viene trasferito nei bit Carry ed Extend
	LSL.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	0 0	X X	31 16 15 0 0 C X Come LSL.B, operando sulla word
	LSL.L	count,Dn Dn,dDn	2	8 + 2N(1/0) 8 + 2N(1/0)			X X	X X	X X	0 0	X X	31 0 C 0 X Come LSL.B, operando sull'intero registro

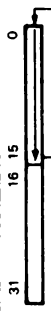
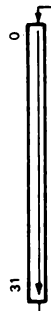
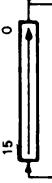
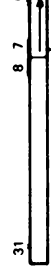
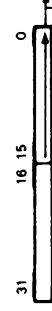
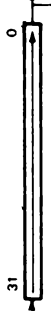
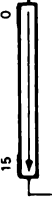



Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	Stato S	Stato X	Stato N	Stato Z	Stato V	Stato C	Operazione eseguita
Shift (continua)	LSR	dadr	2, 4 o 6	9(1/1)+			X	X	X	0	X	Shift logico a destra di un bit di una word di memoria. Uno zero viene inserito nel bit 15; il bit 0 viene trasferito nei bit Carry ed Extend
	LSR.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	0 0	X X	Shift logico a destra del byte di un registro dati. Il numero di shift è specificato direttamente (1-8) o posto in un registro dati (1-63). Ad ogni shift uno zero viene inserito nel bit 7 e il bit 0 viene trasferito nei bit Carry ed Extend
	LSR.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)			X X	X X	X X	0 0	X X	Come LSR.B, operando sulla word
	LSR.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)			X X	X X	X X	0 0	X X	Come LSR.B, operando sull'intero registro
	ROL	dadr	2, 4 o 6	9(1/1)+				X	X	0	X	Rotazione a sinistra di un bit di una word di memoria. Il bit 15 viene trasferito nel bit 0 e nel bit Carry
	ROL.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	Rotazione a sinistra del byte di un registro dati. Il numero di rotazioni viene specificato direttamente (1-8) o posto in un registro dati (1-63). Il bit 7 viene trasferito nel bit 0 e nel bit Carry

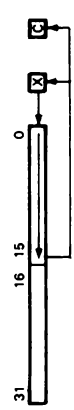
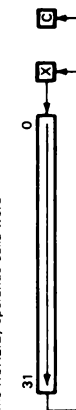
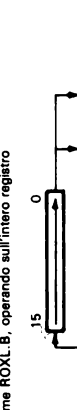
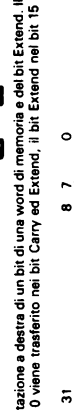



Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Shift (continua)	ROL.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 16 15 0 Come ROL.B, operando sulla word
	ROL.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)				X X	X X	0 0	X X	31 0 Come ROL.B, operando sull'intero registro
	ROR	dadr	2, 4 o 6	9(1/1)+				X	X	0	X	15 0 Rotazione a destra di un bit di una word di memoria. Il bit 0 viene trasferito nel bit 15 e nel bit Carry
	ROR.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 8 7 0 Rotazione a destra del byte di un registro dati. Il numero di rotazioni viene specificato direttamente (1-8) o posto in un registro dati (1-63). Il bit 0 viene trasferito nel bit 7 e nel bit Carry
	ROR.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 16 15 0 Come ROR.B, operando sulla word
	ROR.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)				X X	X X	0 0	X X	31 0 Come ROR.B, operando sull'intero registro
	ROXL	dadr	2, 4 o 6	9(1/1)+				X	X	0	X	15 0 Rotazione a sinistra di un bit di una word di memoria e del bit Extend. Il bit 15 viene trasferito nei bit Extend e Carry; il bit Extend viene trasferito nel bit 0

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	Stato S	Stato X	Stato N	Stato Z	Stato V	Stato C	Operazione eseguita
Shift (continua)	ROXL.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 8 7 0 [C] [X] Rotazione a sinistra del byte di un registro dati e del bit Extend. Il numero di rotazioni viene specificato direttamente (1-8) o posto in un registro dati (1-63). Il bit 7 viene trasferito nei bit Extend e Carry, il bit Extend nel bit 0.
	ROXL. W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 16 15 0 [X] [C] Come ROXL.B, operando sulla word
	ROXL.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)				X X	X X	0 0	X X	31 0 [X] [C] Come ROXL.B, operando sull'intero registro
	ROXR	dadr	2, 4 o 6	9(1/1)+				X	X	0	X	15 0 [X] [C] Rotazione a destra di un bit di una word di memoria e del bit Extend. Il bit 0 viene trasferito nei bit Carry ed Extend, il bit Extend nel bit 15
	ROXR.B	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 8 7 0 [X] [C] Rotazione a destra del byte di un registro dati e del bit Extend. Il numero di rotazioni viene specificato direttamente (1-8) o posto in un registro dati (1-63). Il bit 0 viene trasferito nei bit Extend e Carry, il bit Extend nel bit 7
	ROXR.W	count,Dn Dn,dDn	2 2	6 + 2N(1/0) 6 + 2N(1/0)				X X	X X	0 0	X X	31 16 15 0 [X] [C] Come ROXR.B, operando sulla word
	ROXR.L	count,Dn Dn,dDn	2 2	8 + 2N(1/0) 8 + 2N(1/0)				X X	X X	0 0	X X	31 0 [X] [C] Come ROXR.B, operando sull'intero registro

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Manipolazione di bit	BTST	bitl,Dn	4	10(2/0)					X			$[Z] \leftarrow \overline{[Dn<bitl>]}$
		Dn,dDn	2	6(1/0)					X			$[Z] \leftarrow \overline{[dDn<[Dn]>]}$ Testa lo stato di un bit di un registro dati, riflettendolo nel bit Zero del registro di Stato. Il bit sotto esame può essere specificato direttamente o in un registro dati (bit 0-31 in ambo i casi)
	BTST	bitb,dadr	4, 6 o 8	8(2/0)+					X			$[Z] \leftarrow \overline{[dadr<bitb>]}$
		Dn,dadr	2, 4 o 6	4(1/0)+					X			$[Z] \leftarrow \overline{[dadr<[Dn]>]}$ Testa lo stato di un bit di un byte di memoria, riflettendolo nel bit Zero del registro di Stato. Il bit sotto esame può essere specificato direttamente o in un registro dati (bit 0-31 in ambo i casi)
	BSET	bitl,Dn	4	12(2/0)					X			$[Z] \leftarrow \overline{[Dn<bitl>]}, [Dn<bitl>] \leftarrow 1$
		Dn,dDn	2	8(1/0)					X			$[Z] \leftarrow \overline{[dDn<[Dn]>]}, [dDn<[Dn]> \leftarrow 1$
		bitb,dadr	4, 6 o 8	13(2/1)+					X			$[Z] \leftarrow \overline{[dadr<bitb>]}, [dadr<bitb>] \leftarrow 1$
		Dn,dadr	2, 4 o 6	9(1/1)+					X			$[Z] \leftarrow \overline{[dadr<[Dn]>]}, [dadr<[Dn]>] \leftarrow 1$ Testa lo stato di un bit (come BTST) e lo pone a uno
	BCLR	bitl,Dn	4	14(2/0)					X			$[Z] \leftarrow \overline{[Dn<bitl>]}, [Dn<bitl>] \leftarrow 0$
		Dn,dDn	2	8(1/0)					X			$[Z] \leftarrow \overline{[dDn<[Dn]>]}, [dDn<[Dn]>] \leftarrow 0$
		bitb,dadr	4, 6 o 8	13(2/1)+					X			$[Z] \leftarrow \overline{[dadr<bit\ b>]}, [dadr<bit\ b>] \leftarrow 0$
		Dn,dadr	2, 4 o 6	9(1/1)+					X			$[Z] \leftarrow \overline{[dadr<[Dn]>]}, [dadr<[Dn]>] \leftarrow 0$ Testa lo stato di un bit (come BTST) e lo azzera
	BCHG	bitl,Dn	4	12(2/0)					X			$[Z] \leftarrow \overline{[Dn<bitl>]}, [Dn<bitl>] \leftarrow \overline{[Dn<bitl>]}$
		Dn,dDn	2	8(1/0)					X			$[Z] \leftarrow \overline{[dDn \leftarrow [Dn]>]}, [dDn<[Dn]>], \overline{[dDn<[dDn]>]}$
		bitb,dadr	4, 6 o 8	13(2/1)					X			$[Z] \leftarrow \overline{[dadr<bitb>]}, [dadr<bitb>] \leftarrow \overline{[dadr<bitb>]}$
		Dn,dadr	2, 4 o 6	9(1/1)					X			$[Z] \leftarrow \overline{[dadr<[Dn]>]}, [dadr<[Dn]>] \leftarrow \overline{[dadr<[Dn]>]}$ Testa lo stato di un bit (come BTST) e lo complementa
Stack	MOVE	An,USP	2	4(1/0)								[USP] ← [An] Copia di un registro indirizzi nel Puntatore allo Stack Utente. È un'istruzione privilegiata
	MOVE	USP,An	2	4(1/0)								[An] ← [USP] Copia il contenuto del Puntatore allo Stack Utente in un registro indirizzi. È un'istruzione privilegiata
	LINK	An,d16	4	18(2/2)								[A7] ← [A7] − 2 [[A7]] ← [An] [An] ← [A7] [A7] ← [A7] + d16 Salva sullo stack il contenuto di un registro indirizzi, in cui copia il Puntatore allo Stack corrente; aggiorna quindi il Puntatore allo Stack indirizzandolo oltre l'area temporanea di immagazzinamento in stack
	PEA	jadr	2, 4 o 6	10(1/2)+								[A7] ← [A7] − 2 [[A7]] ← jadr Calcola un indirizzo a 32 bit e lo pone sullo stack[3]

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato T	S	X	N	Z	V	C	Operazione eseguita
Stack (cont.)	UWLK	An	2	12(3/0)								[A7] ← [An] [An] ← [[A7]] [A7] ← [A7] + 2 Copia nel Puntatore allo Stack un registro indirizzi, che viene caricato dallo stack
Interrupt e Trap	CHK	data16,Dn	4	49(6/3), 12(2/0)				X	U	U	U	If [Dn<0-15>] < 0 or [Dn<0-15>] > data16 then [PC] ← CHK vettore interrupt
		Dn,dDn	2	45(5/3), 8(1/0)				X	U	U	U	If [dDn<0-15>] < 0 or [dDn<0-15>] > [Dn<0-15>] then [PC] ← CHK vettore interrupt
		sadr,Dn	2, 4 o 6	45(5/3), 8(1/0)				X	U	U	U	If [Dn<0-15>] <0 or [Dn<0-15>] > [sadr] then [PC] ← CHK vettore interrupt Confronta la word di un reg. con una word (limite sup.), iniziando un processo interrupt di check se fuori dal limite o minore di zero. Il limite superiore, di complemento a due di un intero, viene specificato come dati immediati, in un reg. dati o in una word di mem.[2,3,5]
	TRAP	vector	2	37(4/3)								[A7] ← [A7] - 2 [[A7]] ← [PC] [A7] ← [A7] - 2 [[A7]] ← [SR] [PC] ← vettore Inizia, attraverso il vettore specificato, un processo in modo Exception
	TRAPV		2	37(5/3), 4(1/0)								If Overflow = 1 then TRAP Inizia un processo in modo Exception attraverso il vettore di Overflow se il bit Overflow del registro di Stato è posto a uno
	RTE		2	20(5/0)			X	X	X	X	X	[SR] ← [[A7]], [A7] ← [A7] + 2 [PC] ← [[A7]], [A7] ← [A7] + 2 Ritorno da processo in modo Exception
Stato	MOVE	Dn,CCr	2	12(2/0)			X	X	X	X	X	[SR<0-4>] ← [Dn<0-4] Copia i dati di stato da un registro dati ai codici di condizione
	MOVE	sadr,CCR	2, 4 o 6	12(2/0)+			X	X	X	X	X	[SR<0-4>] [sadr<0-4>] Copia i dati di stato da una locazione di memoria ai codici di condizione. L'indirizzo sorgente è un indirizzo di word[2,3]
	MOVE	data8,CCR	4	16(3/0)			X	X	X	X	X	[SR<0-4>] ← data8<0-4> Muove dati immediati di stato ai codici di condizione
	MOVE	Dn,SR	2	12(2/0)	X	X	X	X	X	X	X	[SR] ← [Dn<0-15>] Copia la word di stato da un registro dati al registro di Stato. È un'istruzione privilegiata
	MOVE	sadr,SR	2, 4 o 6	12(2/0)+	X	X	X	X	X	X	X	[SR] ← [sadr] Copia la word di stato da una locazione di mem. al reg. di Stato. Istruzione privilegiata. L'indirizzo "sorgente" è un indirizzo di word[2,3]
	MOVE	data16,SR	4	16(3/0)	X	X	X	X	X	X	X	[SR] ← data16 Muove una word immediata di stato nel registro di Stato. È una istruzione privilegiata[5]
	MOVE	SR,Dn	2	6(1/0)								[Dn<0-15>] ← [SR] Copia in un registro dati il contenuto del registro di Stato. I bit 16-31 del registro dati restano invariati

Tabella A.2 Sommario del set di istruzioni del microprocessore MC68000 (continua)

	Mnemonici	Operandi	Byte	Cicli di clock	Stato							Operazione eseguita
					T	S	X	N	Z	V	C	
Stato (continua)	MOVE	SR,dadr	2, 4 o 6	9(1/1)+								[dadr] ← [SR] Copia in una locazione di memoria il contenuto del registro di Stato. L'indirizzo "destinazione" è un indirizzo di word[2,3]
	AND.B	data8,SR	4	20(3/0)			X	X	X	X	X	[SR<0-7>] ← [SR<0-7>] ∧ data8 AND logico di un byte di dati immediati al byte del registro di Stato[5]
	AND.W	data16,SR	4	20(3/0)	X	X	X	X	X	X	X	[SR] ← [SR] ∧ data16 AND logico di una word di dati immediati al registro di Stato. È un'istruzione privilegiata
	EOR.B	data8,SR	4	20(3/0)			X	X	X	X	X	[SR<0-7>] ← [SR<0-7>] ⊻ data8 OR esclusivo di un byte di dati immediati al byte del registro di Stato[5]
	EOR.W	data16,SR	4	20(3/0)	X	X	X	X	X	X	X	[SR] ← [SR] ⊻ data16 OR esclusivo di una word di dati immediati al registro di Stato. È un'istruzione privilegiata[5]
	OR.B	data8,SR	4	20(3/0)			X	X	X	X	X	[SR<0-7>] ← [SR<0-7>] ∧ data8 OR logico di un byte di dati immediati al byte del registro di Stato[5]
	OR.W	data16,SR	4	20(3/0)	X	X	X	X	X	X	X	[SR] ← [SR] V data16 OR logico di una word di dati immediati al registro di Stato. È un'istruzione privilegiata[5]
Controlli vari	NOP		2	4(1/0)								Nessuna operazione
	RESET		2	132(1/0)								Reset. È un'istruzione privilegiata
	STOP	data16	4	8(2/0)			X	X	X	X	X	[SR] ← data16 Ferma il processore. È un'istruzione privilegiata

APPENDICE B

TABELLE DEI CODICI OGGETTO DELLE ISTRUZIONI

In questa appendice sono elencati i codici oggetto di tutte le istruzioni dell'MC68000, in ordine alfabetico.

Per le word d'istruzione che non ammettono la possibilità di varianti, i relativi codici oggetto sono rappresentati da quattro cifre esadecimali; ad esempio, 4E71.

Per le word d'istruzione che prevedono delle variazioni in uno dei due byte, il codice oggetto è indicato come una combinazione di variabili minuscole, cifre esadecimali e cifre binarie. Ogni byte di una word d'istruzione è suddiviso in due "nibble" (1 nibble = 4 bit). Se in un nibble compare una sola cifra, si tratterà di una cifra esadecimale. Se vi compaiono, invece, quattro cifre oppure una combinazione di cifre e di variabili minuscole (ad esempio, 1rrr). ciascuna cifra rappresenta un bit.

Notate che alcune variabili minuscole sono utilizzate per rappresentare delle cifre esadecimali, anzichè delle cifre binarie. Quando quattro di questi caratteri esadecimali (ad esempio, xxxx o yyyy) sono utilizzati per rappresentare una word (16 bit), appaiono raggruppati al centro della colonna di 2 byte, contenente quella word.

TEMPI DI ESECUZIONE DELLE ISTRUZIONI

In questa appendice sono indicati i tempi di esecuzione di un'istruzione in cicli di clock. Ogni ciclo = 125 nanosecondi (dove f_{CLK} = 8.0 MHz).

Le abbreviazioni e le notazioni utilizzate nella colonna relativa ai "cicli di clock" hanno il seguente significato:

+ea Tempo aggiuntivo di indirizzamento. È il tempo supplementare necessario all'esecuzione dell'istruzione con gli indirizzamenti più lenti rispetto a quello indiretto a registro.

Modo di Indirizzamento	Cicli supplementari
(An)	0
(An)+	0
-(An)	2
d16(An)	5
d8(An,i)	7
addr-16 bit	5
addr-32 bit	10
label	5
label(i)	7

N	Nelle istruzioni di shift, il numero di spostamenti. Nelle istruzione di trasferimento multiple, il numero dei registri coinvolti.
*	Il primo valore è utilizzato nel caso dell'esecuzione di una diramazione o di una Trap, il secondo in caso di non esecuzione. Con l'istruzione Bcc, il primo di questi numeri è per la versione a due byte (spostamento ad 8 bit) ed il secondo per la versione a quattro byte. Nel caso di DBcc, il primo numero è per diramazione non effettuata per condizione vera ed il secondo per diramazione non effettuata per esaurimento del contatore.
**	Indica il valore massimo
***	Il valore minore è per condizione falsa (byte che contiene tutti uno); il valore maggiore è per condizione vera (byte con tutti zeri).

In questa appendice sono usate le seguenti abbreviazioni:

a	Modo di indirizzamento dell'operando (1 bit) 0 = da registro dati a registro dati 1 = da memoria a memoria
bbb	3 bit di dato immediato. Nelle operazioni su bit, la posizione dei bit (0-7).
bbbbb	Posizioni dei bit (0-31)
ccc	Contatore di shift: 000 = 8 shift 010 = 2 shift 011 = 3 shift 100 = 4 shift 101 = 5 shift 110 = 6 shift 111 = 7 shift
ddd	Registro Destinazione: stessa codifica di rrr.
eeeee	Indirizzo sorgente effettivo (6 bit)

Modo di Ind.	Modo/Reg.	[EXT]
(An)	010rrr	---
(An)+	011rrr	---
-(An)	100rrr	---
d16(An)	101rr	xxxx
d8(An,i)	110rrr	a iii w 000 xx
addr-16 bit	111000	pppp
addr-32 bit	111001	pppp qqqq
label	111010	xxx
label(i)	111011	a iii w 000 xx

[EXT] Una o due word di estensione che possono essere presenti o meno, a seconda del tipo di indirizzamento (vedi la descrizione dei Modi di Indirizzamento).

ffffff Indirizzo destinazione effettivo: analogo ad eeeee, ma senza label o label(i).

gggggg Indirizzo destinazione effettivo, con i campi MODO e REGISTRO scambiati di posto (ad es., (An) = rrr010).

hhhhhh Indirizzo effettivo multi-destinazione: analogo ad ffffff, ma senza (An)+ e -(An).

iii Registro Indice: la stessa codifica di rrr.

jjjjjj Indirizzo effettivo di salto: analogo a eeeeee, ma senza (An)+ e -(An).

kkkk Maschera elenco regsitri per il modo con predecremento, con il seguente formato (un '1' seleziona il registro):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0	A7	A6	A5	A4	A3	A2	A1	A0

mmmm Maschera elenco registri per i modi senza predecremento, con il seguente formato (un '1' seleziona il registro):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

pppp Word di indirizzo a 16 bit o word più significativa di un indirizzo a 32 bit.

qqqq Word meno significativa di un indirizzo a 32 bit.

rrr Registro 000 = D0 o A0
001 = D1 o A1
010 = D2 o A2
011 = D3 o A3
100 = D4 o A4
101 = D5 o A5
110 = D6 o A6
111 = D7 o A7

sss	Registro sorgente: la stessa codifica di rrr
t	Tipo di registro 0 = Dn 1 = An
vvvv	Vettore a 4 bit
w	Grandezza dell'indice. 0 = intero di ordine basso con segno esteso. 1 = valore long word nel registro indice.
xx	Offset di indirizzo a 8 bit
xxxx	Offset di indirizzo a 16 bit
yy	Dato immediato ad 8 bit
yyyy	Dato immediato a 16 bit
zzzz	Word meno significativa di un dato a 32 bit

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte richiesti	Cicli di clock
ABCD	−(SAn),−(dAn)	C dddd1	0 1sss									2	19(3/1)
	sDn,dDn	C dddd1	0 0sss									2	6(1/0)
ADD.B	data8,dadr	0 6	00ff ffff	00	yy	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data8,Dn	0 6	0 0ddd	00	yy							4	8(2/0)
	Dn,dadr	D sss1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	sadr,Dn	D ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sDn,dDn	D ddd0	0 0sss									2	4(1/0)
ADD. L	data32,An	D ddd1	F C	yyyy		zzzz						6	16(3/0)
	data32,dadr	0 6	10ff ffff	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	22(3/2)+
	data32,Dn	0 6	8 0ddd	yyyy		zzzz						6	16(3/0)
	Dn,dadr	D sss1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	rs,An	D ddd1	C tsss									2	8(1/0)
	rs,Dn	D ddd0	8 tsss									2	8(1/0)
	sadr,An	D ddd1	11ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
	sadr,Dn	D ddd0	10ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
ADD. W	data16,An	D ddd0	F C	yyyy								4	8(2/0)
	data16,dadr	0 6	01ff ffff	yyyy		[EXT]		[EXT]				4, 6, o 8	13(2/1)
	data16,Dn	0 6	4 0ddd	yyyy								4	8(2/0)
	Dn,dadr	D ssss1	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(11)+
	rs,An	D ddd0	C tsss									2	8(1/0)
	rs,Dn	D ddd0	4 tsss									2	4(1/0)
	sadr,An	D ddd0	11ee eeee	[EXT]		[EXT]						2, 4, o 6	8(1/0)+
	sadr,Dn	D ddd0	01ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
ADDQ.B	data3,dadr	5 bbb0	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/0)+
	data3,Dn	5 bbb0	0 0ddd									2	4(1/0)
ADDQ.L	data3,An	5 bbb0	8 1ddd									2	8(1/0)
	data3,dadr	5 bbb0	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	data3,Dn	5 bbb0	8 0ddd									2	8(1/0)
ADDQ.W	data3,An	5 bbb0	4 1ddd									2	4(1/0)
	data3,dadr	5 bbb0	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	data3,Dn	5 bbb0	4 0ddd									2	4(1/0)
ADDX.B	−(sAn),−(dAn)	D ddd1	0 1sss									2	19(3/1)
	sDn,dDn	D ddd1	0 0sss									2	4(1/0)
ADDX.L	−(sAn),−(dAn)	D ddd1	8 1sss									2	32(5/2)
	SDn,dDn	D ddd1	8 0sss									2	8(1/0)
ADDX.W	−(sAn),−(dAn)	D ddd1	4 1sss									2	19(3/1)
	sDn,dDn	D ddd1	4 0sss									2	4(1/0)
AND.B	data8,dadr	0 2	00ff ffff	00	yy	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data8,Dn	0 2	0 0ddd	00	yy							4	8(2/0)
	data8,SR	0 2	3 C	00	yy							4	20(3/0)
	Dn,dadr	C sss1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	sadr,Dn	C ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sDn,dDn	C ddd0	0 0sss									2	4(1/0)
AND.L	data32,dadr	0 2	10ff ffff	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	22(3/2)+
	data32,Dn	0 2	8 0ddd	yyyy		zzzz						6	16(3/0)
	Dn,dadr	C sss1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	sadr,Dn	C ddd0	10ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
	sDn,dDn	C ddd0	8 0sss									2	8(1/0)
AND.W	data16,dadr	0 2	01ff ffff	yyyy		[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data16,Dn	0 2	4 0ddd	yyyy								4	8(2/0)
	data16,SR	0 2	7 C	yyyy								4	20(3/0)
	Dn,dadr	C sss1	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)
	sadr,Dn	C ddd0	01ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte richiesti	Cicli di clock
	sDn,dDn	C ddd0	4 0sss									2	4(1/0)
ASL	dadr	E 1	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
ASL.B	count,Dn	E ccc1	0 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	2 0ddd									2	6 + 2N(1/0)
ASL.L	count,Dn	E ccc1	8 0ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr1	A 0ddd									2	8 + 2N(1/0)
ASL.W	count,Dn	E ccc1	4 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	6 0ddd									2	6 + 2N(1/0)
ASR	dadr	E 0	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
ASR.B	count,Dn	E ccc0	0 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	2 0ddd									2	6 + 2N(1/0)
ASR.L	count,Dn	E ccc0	8 0ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr0	A 0ddd									2	8 + 2N(1/0)
ASR.W	count,Dn	E ccc0	4 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	6 0ddd									2	6 + 2N(1/0)
BCC	label	6 4	0 0	xxxx								4	10, 12(2/0)
	label	6 4	x x									2	10, 18(1/0)
BCHG	bitb,dadr	0 8	01ff ffff	00	0 0bbb	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	bitl,Dn	0 8	4 0ddd	00	000b bbbb							4	12(2/0)
	Dn,dadr	0 rrr1	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn,dDn	0 rrr1	4 0ddd									2	8(1/0)
BCLR	bitb,dadr	0 8	10ff ffff	00	0 0bbb	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	bitl,Dn	0 8	8 0ddd	00	000b bbbb							4	14(2/0)
	Dn,dadr	0 rrr1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn,dDn	0 rrr1	8 0ddd									2	8(1/0)
BCS	label	6 5	0 0	xxxx								4	10, 12(2/0)
	label	6 5	x x									2	10, 18(1/0)
BEQ	label	6 7	0 0	xxxx								4	10, 12(2/0)
	label	6 7	x x									2	10, 8(1/0)
BGE	label	6 C	0 0	xxxx								4	10, 12(2/0)
	label	6 C	x x									2	10, 8(1/0)
BGT	label	6 E	0 0	xxxx								4	10, 12(2/0)
	label	6 E	x x									2	10, 8(1/0)
BHI	label	6 2	0 0	xxxx								4	10, 12(2/0)
	label	6 2	0 0									2	10, 8(1/0)
BLE	label	6 F	0 0	xxxx								4	10, 12(2/0)
	label	6 F	x x									2	10, 8(1/0)
BLS	label	6 3	0 0	xxxx								4	10, 12(2/0)
	label	6 3	x x									2	10, 8(1/0)
BLT	label	6 D	0 0	xxxx								4	10, 12(2/0)
		6 D	x x									2	10, 8(1/0)
BMI	label	6 B	0 0	xxxx								4	10, 12(2/0)
	label	6 B	x x									2	10, 8(1/0)
BNE	label	6 6	0 0	xxxx								4	10, 12(2/0)
	label	6 6	x x									2	10, 8(1/0)
BPL	label	6 A	0 0	xxxx								4	10, 12(2/0)
	label	6 A	x x									2	10, 8(1/0)
BRA	label	6 0	0 0	xxxx								4	10(2/0)
	label	6 0	x x									2	10(1/0)
BSET	bitb,dadr	0 8	11ff ffff	00	0 0bbb	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	bitl,Dn	0 8	C 0ddd	00	000b bbbb							4	12(2/0)
	Dn,dadr	0 rrr1	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn,dDn	0 rrr1	C 0ddd									2	8(1/0)

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte rich.	Cicli di clock
BSR	label	6 1	0 0	xxxx								4	10, 12(2/0)
	label	6 1	x x									2	10, 8(1/0)
BTST	bitb,dadr	0 8	00ff ffff	00	0 0bbb	[EXT]		[EXT]				4, 6, o 8	8(2/0)+
	bitl,Dn	0 8	0 0ddd	00	000b bbbb							4	10(2/0)
	Dn,dadr	0 rrr1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	4(1/0)
	Dn,dDn	0 rrr1	0 0ddd									2	6(1/0)
BVC	label	6 8	0 0	xxxx								4	10, 12(2/0)
	label	6 8	x x									2	10, 8(1/0)
BVS	label	6 9	0 0	xxxx								4	10, 12(2/0)
		6 9	x x									2	10, 8(1/0)
CHK	data16,Dn	4 ddd1	B C	yyyy								4	49(6/3),12(2/0)
	Dn,dDn	4 ddd1	8 0rrr									2	45(5/3),8(1/0)
	sadr,Dn	4 ddd1	10ee eeee	[EXT]		[EXT]						2, 4, o 6	45(5/3)+,81/0)
CLR.B	dadr	4 2	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	4 2	0 0ddd									2	4(1/0)
CLR.L	dadr	4 2	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	Dn	4 2	8 0ddd									2	6(1/0)
CLR.W	dadr	4 2	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)
	Dn	4 2	4 0ddd									2	4(1/0)
CMP. B	data8,dadr	0 C	00ff ffff	00	yy	[EXT]		[EXT]				4, 6, o 8	8(2/0)
	data8,Dn	0 C	0 0ddd	00	yy							4	8(2/0)
	sadr,Dn	B ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sDn,dDn	B ddd0	0 0sss									2	4(1/0)
CMP. L	data32,An	B ddd1	F C	yyyy		zzzz						6	14(3/0)
	data32,dadr	0 C	10ff ffff	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	12(3/0)+
	data32,Dn	0 C	8 0ddd	yyyy		zzzz						6	14(3/0)
	rs,An	B ddd1	C tsss									2	6(1/0)
	rs,Dn	B ddd0	8 tsss									2	6(1/0)
	sadr,An	B ddd1	11ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
	sadr,Dn	B ddd0	10ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
CMP. W	data16,An	B ddd0	F C	yyyy								4	8(2/0)
	data16,dadr	0 C	01ff ffff	yyyy		[EXT]		[EXT]				4, 6, o 8	8(2/0)+
	data16,Dn	0 C	4 0ddd	yyyy								4	8(2/0)
	rs,An	B ddd0	C tsss									2	6(1/0)
	rs,Dn	B ddd0	4 tsss									2	4(1/0)
	sadr,An	B ddd0	11ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
	sadr,Dn	B ddd0	01ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
CMPM. B	(sAn)+,(dAn)+	B ddd1	0 1sss									2	12(3/0)
CMPM. L	(sAn)+,(dAn)+	B ddd1	8 1sss									2	20(5/0)
CMPM. W	(sAn)+,(dAn)+	B ddd1	4 1sss									2	12(3/0)
DBCC	Dn,label	5 4	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBCS	Dn,label	5 5	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBEQ	Dn,label	5 7	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBF	Dn,label	5 1	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBGE	Dn,label	5 C	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBGT	Dn,label	5 E	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBHI	Dn,label	5 2	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBLE	Dn,label	5 F	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBLS	Dn,label	5 3	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBLT	Dn,label	5 D	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBMI	Dn,label	5 B	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBNE	Dn,label	5 6	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DBPL	Dn,label	5 A	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte rich.	Cicli di clock
DBRA	Dn,label	(same as DBF)											
DBT	Dn,label	5 0	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DVC	Dn,label	5 8	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DVS	Dn,label	5 9	C 1rrr	xxxx								4	12(2/0),10(2/0),14(3/0)
DIVS	data16,Dn	8 ddd1	F C	yyyy								4	<162(2/0)
	sadr,Dn	8 ddd1	11ee eeee	[EXT]		[EXT]						2, 4, o 6	<158(1/0)+
	sDn,dDn	8 ddd1	C 0sss									2	<158(1/0)
DIVU	data16,Dn	8 ddd0	F C	yyyy								4	<148(2/0)
	sadr,Dn	8 ddd0	11ee eeee	[EXT]		[EXT]						2, 4, o 6	<140(1/0)+
	sDn,dDn	8 ddd0	C 0sss									2	<140(1/0)
EOR.B	data8,dadr	0 A	00ff ffff	00	yy	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data8,Dn	0 A	0 0ddd	00	yy							4	8(2/0)
	data8,SR	0 A	3 C	00	yy							4	20(3/0)
	Dn,dadr	B sss1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	sDn,dDn	B sss1	0 0ddd									2	4(1/0)
EOR. L	data32,dadr	0 A	10ff ffff	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	22(3/2)+
	data32,Dn	0 A	8 0ddd	yyyy		zzzz						6	16(3/0)
	Dn,dadr	B sss1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	sDn,dDn	B sss1	8 0ddd									2	8(1/0)
EOR. W	data16,dadr	0 A	01ff ffff	yyyy		[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data16,Dn	0 A	4 0ddd	yyyy								4	8(2/0)
	data16,Sr	0 A	7 C	yyyy								4	20(3/0)
	Dn,dadr	B sss1	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	sDn,dDn	B sss1	4 0ddd									2	4(1/0)
EXG	An,An	C sss1	4 1ddd									2	6(1/0)
	An,Dn	C sss1	8 1ddd									2	6(1/0)
	Dn,An	(same as An,Dn)											
	Dn,Dn	C sss1	4 0ddd									2	6(1/0)
EXT.L	Dn	4 8	C 0ddd									2	4(1/0)
EXT.W	Dn	4 8	8 0ddd									2	4(1/0)
JMP	jadr	4 E	11jj jjjj	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
JSR	jadr	4 E	10jj jjjj	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
LEA	jadr,An	4 ddd1	11jj jjjj	[EXT]		[EXT]						2, 4, o 6	2(0/0)+
LINK	An,d16	4 E	5 0rrr			xxxx						4	18(2/2)
LSL	dadr	E 3	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
LSL.B	count,Dn	E ccc1	0 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	2 1ddd									2	6 + 2N(1/0)
LSL.L	count,Dn	E ccc1	8 1ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr1	A 1ddd									2	8 + 2N(1/0)
LSL.W	count,Dn	E ccc1	4 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	6 1ddd									2	6 + 2N(1/0)
LSR	dadr	E 2	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
LSR.B	count,Dn	E ccc0	0 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	2 1ddd									2	6 + 2N(1/0)
LSR.L	count,Dn	E ccc0	8 1ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr0	A 1ddd									2	8 + 2N(1/0)
LSR.W	count,Dn	E ccc0	4 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	6 1ddd									2	6 + 2N(1/0)
MOVE	An,USP	4 E	6 0sss									2	4(1/0)
	data8,CCR	4 4	F C	00	yy							4	16(3/0)
	data16,SR	4 6	F C	yyyy								4	16(3/0)
	Dn,CCR	4 4	C 0sss									2	12(2/0)
	Dn,SR	4 6	C 0sss									2	12(2/0)

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte richiesti	Cicli di clock
	sadr,CCR	4 4	11ee eeee	[EXT]		[EXT]						2, 4, o 6	12(2/0)+
	sadr,SR	4 6	11ee eeee	[EXT]		[EXT]						2, 4, o 6	12(2/0)+
	SR,dadr	4 0	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	SR,Dn	4 0	C 0ddd									2	6(1/0)
	USP,An	4 E	6 1sss									2	4(1/0)
MOVE.8	data8,Dn	1 ddd0	3 C	00	yy							4	8(2/0)
	data8,dadr	1 gggg	gg11 C	00	yy	[EXT]		[EXT]				4, 6, o 8	9(1/1)+
	Dn,dadr	1 gggg	gg00 0sss	[EXT]		[EXT]						2, 4, o 6	5(0/1)+
	sDn,dDn	1 ddd0	0 0sss									2	4(1/0)
	sadr,dadr	1 gggg	ggee eeee	[EXT$_s$]		[EXT$_s$]		[EXT$_d$]		[EXT$_d$]		2, 4, 6, 8 o 10	5(1/1)+
	sadr,Dn	1 ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
MOVE. L	An,dadr	2 gggg	gg00 1sss	[EXT]		[EXT]						2, 4, o 6	10(0/2)+
	data32,An	2 ddd0	7 C	yyyy		zzzz						6	12(3/0)
	data32,dadr	2 gggg	gg11 C	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	18(2/2)+
	data32,Dn	2 ddd0	3 C	yyyy		zzzz						6	12(3/0)
	Dn,dadr	2 gggg	gg00 0sss	[EXT]		[EXT]						2, 4, o 6	10(0/2)+
	rs,An	2 ddd0	4 tsss									2	4(1/0)
	rs,Dn	2 ddd0	0 tsss									2	4(1/0)
	sadr,An	2 ddd0	01ee eeee	[EXT]		[EXT]						2, 4, o 6	8(2/0)+
	sadr,dadr	2 gggg	ggee eeee	[EXT$_s$]		[EXT$_s$]		[EXT$_d$]		[EXT$_d$]		2, 4, 6, 8, o 10	14(1/2)+
	sadr,Dn	2 ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
MOVE. W	An,dadr	3 gggg	gg00 1sss	[EXT]		[EXT]						2, 4, o 6	5(0/1)+
	data16,An	3 ddd0	7 C	yyyy								4	8(2/0)
	data16,dadr	3 gggg	gg11 C	yyyy		[EXT]		[EXT]				4, 6, o 8	9(1/1)+
	data16,Dn	3 ddd0	3 C	yyyy								4	8(2/0)
	Dn,dadr	3 gggg	gg00 0sss	[EXT]		[EXT]						2, 4, o 6	5(0/1)+
	rs,An	3 ddd0	4 tsss									2	4(1/0)
	rs,Dn	3 ddd0	0 tsss									2	4(1/0)
	sadr,An	3 ddd0	01ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sadr,dadr	3 gggg	ggee eeee	[EXT$_s$]		[EXT$_s$]		[EXT$_d$]		[EXT$_d$]		2, 4, 6, 8, o 10	5(0/1)+
	sadr,Dn	3 ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
MOVEM. L	(An)+,reg-list	4 C	E 0sss	mmmm								4	8 + 8n(2 + 2n/0)
	jadr,reg-list	4 C	11jj jjjj	mmmm		[EXT]		[EXT]				4, 6, o 8	8 + 8n(2 + 2n/0)+
	reg-list,−(An)	4 8	E 0ddd	kkkk								4	4 + 10n(1/n)
	reg-list,madr	4 8	11hh hhhh	mmmm		[EXT]		[EXT]				4, 6, o 8	4 + 10n(1/n)+
MOVEM. W	(An)+,reg-list	4 C	A 0sss	mmmm								4	8 + 4n(2 + n/0)
	jadr,reg-list	4 C	10jj jjjj	mmmm		[EXT]		[EXT]				4, 6, o 8	8 + 4n(2 + n/0)+
	reg-list,−(An)	4 8	A 0ddd	kkkk								4	4 + 5n(1/n)
	reg-list,madr	4 8	10hh hhhh	mmmm		[EXT]		[EXT]				4, 6, o 8	4 + 5n(1/n)+
MOVEP. L	d16(An),Dn	0 ddd1	4 1sss	xxxx								4	24(6/0)
	Dn,d16(An)	0 sss1	C 1ddd	xxxx								4	28(2/4)
MOVEP. W	d16(An),Dn	0 ddd1	0 1sss	xxxx								4	16(4/0)
	Dn,d16(An)	0 sss1	8 1ddd	xxxx								4	18(2/2)
MOVEQ	data8,Dn	7 ddd0	y y									2	4(1/0)
MULS	data16,Dn	C ddd1	F C	yyyy								4	<74(2/0)
	sadr,Dn	C ddd1	11ee eeee	[EXT]		[EXT]						2, 4, o 6	<70(1/0)+
	sDn,dDn	C ddd1	C 0sss									2	<70(1/0)
MULU	data16,Dn	C ddd0	F C	yyyy								4	<74(2/0)
	sadr,Dn	C ddd0	11ee eeee	[EXT]		[EXT]						2, 4, o 6	<70(1/0)+
	sDn,dDn	C ddd0	C 0sss									2	<70(1/0)
NBCD	dadr	4 8	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	4 8	0 0ddd									2	6(1/0)
NEG. B	dadr	4 4	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte richiesti	Cicli di clock
	Dn	4 4	0 0ddd									2	4(1/0)
NEG.L	dadr	4 4	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	Dn	4 4	8 0ddd									2	6(1/0)
NEG.W	dadr	4 4	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)
	Dn	4 4	4 0ddd									2	4(1/0)
NEGX.B	dadr	4 0	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	4 0	0 0ddd									2	4(1/0)
NEGX.L	dadr	4 0	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	Dn	4 0	8 0ddd									2	6(1/0)
NEGX.W	dadr	4 0	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	4 0	4 0ddd									2	4(1/0)
NOP		4 E	7 1									2	4(1/0)
NOT.B	dadr	4 6	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	4 6	0 0ddd									2	4(1/0)
NOT.L	dadr	4 6	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	Dn	4 6	8 0ddd									2	6(1/0)
NOT. W	dadr	4 6	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	4 6	4 0ddd									2	6(1/0)
OR. B	data8,dadr	0 0	00ff ffff	00	yy	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data8,Dn	0 0	0 0ddd	00	yy							4	8(2/0)
	data8,SR	0 0	3 C	00	yy							4	20(3/0)
	Dn,dadr	8 sss1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	sadr,Dn	8 ddd0	00eeeeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sDn,dDn	8 ddd0	0 0sss									2	4(1/0)
OR. L	data32,dadr	0 0	10ff ffff	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	22(3/2)+
	data32,Dn	0 0	8 0ddd	yyyy		zzzz						6	16(3/0)
	Dn,dadr	8 sss1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	sadr,Dn	8 ddd0	10eeeeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
	sDn,dDn	8 ddd0	8 0sss									2	8(1/0)
OR. W	data16,dadr	0 0	01ff ffff	yyyy		[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data16,Dn	0 0	4 0ddd	yyyy								4	8(2/0)
	data16,SR	0 0	7 C	yyyy								4	20(3/0)
	Dn,dadr	8 ss1	10ff ffff			[EXT]		[EXT]				2, 4, o 6	9(1/1)+
	sadr,Dn	8 ddd0	01eeeeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sDn,dDn	8 ddd0	4 0sss									2	4(1/0)
PEA	jadr	4 8	01jj jjjj	[EXT]		[EXT]						2, 4, o 6	10(1/2)+
RESET		4 E	7 0									2	132(1/0)
ROL	dadr	E 7	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
ROL.B	count,Dn	E ccc1	1 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	3 1ddd									2	6 + 2N(1/0)
ROL.L	count,Dn	E ccc1	9 1ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr1	B 1ddd									2	8 + 2N(1/0)
ROL.W	count,Dn	E ccc1	5 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	7 1ddd									2	6 + 2N(1/0)
ROR	dadr	E 6	11ff ffff	[EXT]		[EXT]						2, 4 o 6	9(1/1)+
ROR.B	count,Dn	E ccc0	1 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	3 1ddd									2	6 + 2N(1/0)
ROR.L	count,Dn	E ccc0	9 1ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr0	B 1ddd									2	8 + 2N(1/0)
ROR.W	count,Dn	E ccc0	5 1ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	7 1ddd									2	6 + 2N(1/0)
ROXL	dadr	E 5	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
ROXL.B	count,Dn	E ccc1	1 0ddd									2	6 + 2N(1/0)

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte richiesti	Cicli di clock
	Dn,dDn	E rrr1	3 0ddd									2	6 + 2N(1/0)
ROXL.L	count,Dn	E ccc1	9 0ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr1	B 0ddd									2	8 + 2N(1/0)
ROXL.W	count,Dn	E ccc1	5 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr1	7 0ddd									2	6 + 2N(1/0)
ROXR	dadr	E 4	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
ROXR.B	count,Dn	E ccc0	1 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	3 0ddd									2	6 + 2N(1/0)
ROXR.L	count,Dn	E ccc0	4 0ddd									2	8 + 2N(1/0)
	Dn,dDn	E rrr0	B 0ddd									2	8 + 2N(1/0)
ROXR.W	count,Dn	E ccc0	5 0ddd									2	6 + 2N(1/0)
	Dn,dDn	E rrr0	7 0ddd									2	6 + 2N(1/0)
RTE		4 E	7 3									2	20(5/0)
RTR		4 E	7 7									2	20(5/0)
RTS		4 E	7 5									2	16(4/0)
SBCD	−(sAn),−(dAn)	8 ddd1	0 1sss									2	19(3/1)
	sDn,dDn	8 ddd1	0 0sss									2	6(1/0)
SCC	dadr	5 4	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 4	C 0ddd									2	6, 4(1/0)
SCS	dadr	5 5	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 5	C 0ddd									2	6, 4(1/0)
SEQ	dadr	5 7	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 7	C 0ddd									2	6, 4, (1/0)
SF	dadr	5 1	1fff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 1	C 0ddd									2	6, 4(1/0)
SGE	dadr	5 C	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 C	C 0ddd									2	6, 4(1/0)
SGT	dadr	5 E	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)
	Dn	5 E	C 0ddd									2	6, 4(1/0)
SHI	dadr	5 2	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 2	C 0ddd									2	6, 4,(1/0)
SLE	dadr	5 F	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 F	C 0ddd									2	6, 4(1/0)
SLS	dadr	5 3	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 3	C 0ddd									2	6, 4(1/0)
SLT	dadr	5 D	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 D	C 0ddd									2	6, 4(1/0)
SMI	dadr	5 B	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 B	C 0ddd									2	6, 4(1/0)
SNE	dadr	5 6	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 6	C 0ddd									2	6, 4(1/0)
SPL	dadr	5 A	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 A	C 0ddd									2	6, 4(1/0)
ST	dadr	5 0	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 0	C 0ddd									2	6, 4(1/0)
STOP	data16	4 E	7 2	yyyy								4	8(2/0)
SUB. B	data8,dadr	0 4	00ff ffff	00	yy	[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data8,Dn	0 4	0 0ddd	00	yy							4	8(2/0)
	Dn,dadr	9 sss1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	sadr,Dn	9 ddd0	00ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	sDn,dDn	9 ddd0	0 0sss									2	4(1/0)
SUB. L	data32,An	9 ddd1	F C	yyyy		zzzz						6	16(3/0)
	data32,dadr	0 4	10ff ffff	yyyy		zzzz		[EXT]		[EXT]		6, 8, o 10	22(3/2)+

Tabella B.1 Codici oggetto delle istruzioni del microprocessore MC68000 (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte richiesti	Cicli di clock
	data32,Dn	0 4	8 0ddd	yyyy		zzzz						6	16(3/0)
	Dn,dadr	9 sss1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	rs,An	9 ddd1	C tsss									2	8(1/0)
	rs,Dn	9 ddd0	8 tsss									2	8(1/0)
	sadr,An	9 ddd1	11ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
	sadr,Dn	9 ddd0	10ee eeee	[EXT]		[EXT]						2, 4, o 6	6(1/0)+
SUB.W	data16,An	9 ddd0	F C	yyyy								4	8(2/0)
	data16,dadr	0 4	01ff ffff	yyyy		[EXT]		[EXT]				4, 6, o 8	13(2/1)+
	data16,Dn	0 4	4 0ddd	yyyy								4	8(2/0)
	Dn,dadr	9 sss1	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	rs,An	9 ddd0	C tsss									2	8(1/0)
	rs,Dn	9 ddd0	4 tsss									2	4(1/0)
	sadr,An	9 ddd0	11ee eeee	[EXT]		[EXT]						2, 4, o 6	8(1/0)+
	sadr,Dn	9 ddd0	01ee eeee	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
SUBQ. B	data3,dadr	5 bbb1	00ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	data3,Dn	5 bbb1	0 0ddd									2	4(1/0)
SUBQ.L	data3,An	5 bbb1	8 1ddd									2	8(1/0)
	data3,dadr	5 bbb1	10ff ffff	[EXT]		[EXT]						2, 4, o 6	14(1/2)+
	data3,Dn	5 bbb1	8 0ddd									2	8(1/0)
SUBQ.W	data3,An	5 bbb1	4 1ddd									2	4(1/0)
	data3,dadr	5 bbb1	01ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	data3,Dn	5 bbb1	4 0ddd									2	4(1/0)
SUBX.B	–(sAn), –(dAn)	9 ddd1	0 1sss									2	19(3/1)
	sDn, dDn	9 ddd1	0 0sss									2	4(1/0)
SUBX.L	–(sAn),–(dAn)	9 ddd1	8 1sss									2	32(5/2)
	sDn,dDn	9 ddd1	8 0sss									2	8(1/0)
SUBX.W	–(sAn),–(dAn)	9 ddd1	4 1sss									2	19(3/1)
	sDn,dDn	9 ddd1	4 0sss									2	4(1/0)
SVC	dadr	5 8	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 8	C 0ddd									2	6, 4(1/0)
SVS	dadr	5 9	11ff ffff	[EXT]		[EXT]						2, 4, o 6	9(1/1)+
	Dn	5 9	C 0ddd									2	6, 4(1/0)
SWAP	Dn	4 8	4 0rrr									2	4(1/0)
TAS	dadr	4 A	11ff ffff	[EXT]		[EXT]						2, 4, o 6	11(1/1)+
	Dn	4 A	C 0rrr									2	4(1/0)
TRAP	vector	4 E	4 0vvv									2	36(4/3)
TRAPV		4 E	7 6									2	37(5/3),4(1/0)
TST.B	dadr	4 A	00ff ffff	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	Dn	4 A	0 0rrr									2	4(1/0)
TST.L	dadr	4 A	10ff ffff	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	Dn	4 A	8 0rrr									2	4(1/0)
TST.W	dadr	4 A	01ff ffff	[EXT]		[EXT]						2, 4, o 6	4(1/0)+
	Dn	4 A	4 0rrr									2	4(1/0)
UNLK	An	4 E	5 1rrr									2	12(3/0)

APPENDICE C

CODICI OGGETTO DELLE ISTRUZIONI ELENCATI IN ORDINE NUMERICO

Questa appendice elenca i codici oggetto delle istruzioni in ordine numerico ascendente. Per le indicazioni riguardanti il formato, le abbreviazioni e le notazioni utilizzate in questa appendice, consultate le prime tre pagine dell'Appendice B.

Tabella B.2 Codici oggetto delle istruzioni del microprocessore MC68000 in ordine numerico

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
OR.B	data8,Dn	00	0 0ddd	00	yy						
OR.B	data8,dadr	00	00ff ffff	00	yy	[EXT]		[EXT]			
OR.B	data8,SR	00	3 C	00	yy						
OR.W	data16,Dn	00	4 0ddd	yyyy							
OR.W	data16,dadr	00	01ff ffff	yyyy		[EXT]		[EXT]			
OR.W	data16,SR	00	7 C	yyyy							
OR.L	data32,Dn	00	8 0ddd	yyyy		zzzz					
OR. L	data32,dadr	00	10ff ffff	yyyy		zzzz		[EXT]		[EXT]	
BTST	Dn,dDn	0 rrr1	0 0ddd								
MOVEP.W	d16(An),Dn	0 ddd1	0 1sss								
BTST	Dn,dadr	0 rrr1	00ff ffff	[EXT]		[EXT]					
BCHG	Dn,dDn	0 rrr1	A 0ddd								
MOVEP.L	d16(An),Dn	0 ddd1	A 1sss	xxxx							
BCHG	Dn,dadr	0 rrr1	01ff ffff	[EXT]		[EXT]					
BCLR	Dn,dDn	0 rrr1	8 0ddd								
MOVEP.W	Dn,d16(An)	0 sss1	8 1ddd	xxxx							
BCLR	Dn,dadr	0 rrr1	10ff ffff	[EXT]		[EXT]					
BSET	Dn,dDn	0 rrr1	C 0ddd								
MOVEP.L	Dn,d16(An)	0 sss1	C 1ddd	xxxx							
BSET	Dn,dadr	0 rrr1	11ff ffff	[EXT]		[EXT]					
AND.B	data8,Dn	02	0 0ddd	00	yy						
AND.B	data8,dadr	02	00ff ffff	00	yy	[EXT]		[EXT]			
AND.B	data8,SR	02	3 C	00	yy						
AND.W	data16,Dn	02	4 0ddd	yyyy							
AND.W	data16,dadr	02	01ff ffff	yyyy		[EXT]		[EXT]			
AND.W	data16,SR	02	7 C	yyyy							
AND.L	data32,Dn	02	8 0ddd	yyyy		zzzz					
AND.L	data32,dadr	02	10ff ffff	yyyy		zzzz		[EXT]		[EXT]	
SUB.B	data8,Dn	04	0 0ddd	00	yy						
SUB.B	data8,dadr	04	00ff ffff	00	yy	[EXT]		[EXT]			
SUB.W	data16,Dn	04	4 0ddd	yyyy							
SUB.W	data16,dadr	04	01ff ffff	yyyy		[EXT]		[EXT]			
SUB.L	data32,Dn	04	8 0ddd	yyyy		zzzz					
SUB.L	data32,dadr	04	10ff ffff	yyyy		zzzz		[EXT]		[EXT]	
ADD.B	data8,Dn	06	0 0ddd	00	yy						
ADD.B	data8,dadr	06	00ff ffff	00	yy	[EXT]		[EXT]			
ADD.W	data16,Dn	06	4 0ddd	yyyy							
ADD.W	data16,dadr	06	01ff ffff	yyyy		[EXT]		[EXT]			
ADD.L	data32,Dn	06	8 0ddd	yyyy		zzzz					
ADD.L	data32,dadr	06	10ff ffff	yyyy		zzzz		[EXT]		[EXT]	
BTST	bitl,Dn	08	0 0ddd	00	000b bbbb						
BTST	bitb,dadr	08	00ff ffff	00	0 0bbb	[EXT]		[EXT]			
BCHG	bitl,Dn	08	4 0ddd	00	000b bbbb						
BCHG	bitb,dadr	08	01ff ffff	00	0 0bbb	[EXT]		[EXT]			
BCLR	bitl,Dn	08	8 0ddd	00	000b bbbb						
BCLR	bitb,dadr	08	10ff ffff	00	0 0bbb	[EXT]		[EXT]			
BSET	bitl,Dn	08	C 0ddd	00	000b bbbb						
BSET	bitb,dadr	08	11ff ffff	00	0. 0bbb	[EXT]		[EXT]			
EOR.B	data8,Dn	0A	0 0ddd	00	yy						
EOR.B	data8,dadr	0A	00ff ffff	00	yy	[EXT]		[EXT]			
EOR.B	data8,SR	0A	3 C		yy						
EOR.W	data16,Dn	0A	4 0ddd	yyyy							
EOR.W	data16,dadr	0A	01ff ffff	yyyy		[EXT]		[EXT]			
EOR.W	data16,SR	0A	7 C	yyyy							
EOR.L	data32,Dn	0A	8 0ddd	yyyy		zzzz					
EOR.L	data32,dadr	0A	10ff ffff	yyyy		zzzz		[EXT]		[EXT]	
CMP.B	data8,Dn	0C	0 0ddd	00	yy						
CMP.B	data8,dadr	0C	00ff ffff	00	yy	[EXT]		[EXT]			
CMP.W	data16,Dn	0C	4 0ddd	yyyy							
CMP.W	data16,dadr	0C	01ff ffff	yyyy		[EXT]		[EXT]			
CMP.L	data32,Dn	0C	8 0ddd	yyyy		zzzz					
CMP.L	data32,dadr	0C	10ff ffff	yyyy		zzzz		[EXT]		[EXT]	
MOVE.B	sDn,dDn	1 ddd0	0 0sss								
MOVE.B	sadr,Dn	1 ddd0	00ee eeee	[EXT]		[EXT]					
MOVE.B	data8,Dn	1 ddd0	3 C	00	yy						
MOVE.B	Dn,dadr	1 gggg	gg00 0sss	[EXT]		[EXT]					
MOVE.B	sadr,dadr	1 gggg	ggee eeee	[EXT$_s$]		[EXT$_s$]		[EXT$_d$]		[EXT$_d$]	
MOVE.B	data8,dadr	1 gggg	gg11 C	00	yy	[EXT]		[EXT]			
MOVE.L	rs,Dn	2 ddd0	0000 tsss								
MOVE.L	sadr,Dn	2 ddd0	00ee eeee	[EXT]		[EXT]					
MOVE.L	data32,Dn	2 ddd0	3 C	yyyy		zzzz					
MOVE.L	rs,An	2 ddd0	0100 tsss								

Tabella B.2 Codici oggetto delle istruzioni del microprocessore MC68000 in ordine numerico (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
MOVE.L	data32,An	2ddd0	7 C	yyyy		zzzz					
MOVE.L	rs,dadr	2gggg	gg00tss	[EXT]		[EXT]					
MOVE.L	sadr,dadr	2gggg	ggee eeee	$[EXT_s]$		$[EXT_s]$		$[EXT_d]$		$[EXT_d]$	
MOVE.L	data32,dadr	2gggg	gg11 C	yyyy		zzzz		[EXT]		[EXT]	
MOVE.W	rs,Dn	3ddd0	0 tsss								
MOVE.W	sadr,Dn	3ddd0	00ee eeee	[EXT]		[EXT]					
MOVE.W	data16,Dn	3ddd0	3 C	yyyy							
MOVE.W	rs,An	3ddd0	4 tsss								
MOVE.W	sadr,An	3ddd0	01ee eeee	[EXT]		[EXT]					
MOVE.W	data16,An	3ddd0	7 C	yyyy							
MOVE.W	rs,dadr	3gggg	gg00 tsss	[EXT]		[EXT]					
MOVE.W	sadr,dadr	3gggg	ggee eeee	$[EXT_s]$		$[EXT_s]$		$[EXT_d]$		$[EXT_d]$	
MOVE.W	data16,dadr	3gggg	gg11 C	yyyy		[EXT]		[EXT]			
NEGX.B	Dn	40	0 0ddd								
NEGX.B	dadr	40	00ff ffff	[EXT]		[EXT]					
NEGX.W	Dn	40	4 0ddd								
NEGX.W	dadr	40	01ff ffff	[EXT]		[EXT]					
NEGX.L	Dn	40	8 0ddd								
NEGX.L	dadr	40	10ff ffff	[EXT]		[EXT]					
MOVE	SR,Dn	40	C 0ddd								
MOVE	SR,dadr	40	11ff ffff	[EXT]		[EXT]					
CHK	Dn,dDn	4ddd1	8 0rrr								
CHK	sadr,Dn	4ddd1	10ee eeee	[EXT]		[EXT]					
CHK	data16,Dn	4ddd1	B C	yyyy							
LEA	jadr,An	4ddd1	11jj jjjj	[EXT]		[EXT]					
CLR.B	Dn	42	0 0ddd								
CLR.B	dadr	42	00ff ffff	[EXT]		[EXT]					
CLR.W	Dn	42	4 0ddd								
CLR.W	dadr	42	01ff ffff	[EXT]		[EXT]					
CLR.L	Dn	42	8 0ddd								
CLR.L	dadr	42	10ff ffff	[EXT]		[EXT]					
NEG.B	Dn	44	0 0ddd								
NEG.B	dadr	44	00ff ffff	[EXT]		[EXT]					
NEG.W	Dn	44	4 0ddd								
NEG.W	dadr	44	01ff ffff	[EXT]		[EXT]					
NEG.L	Dn	44	8 0ddd								
NEG.L	dadr	44	10ff ffff	[EXT]		[EXT]					
MOVE	Dn,CCR	44	C 0sss								
MOVE	sadr,CCR	44	11ee eeee	[EXT]		[EXT]					
MOVE	data8,CCR	44	F C	00	yy						
NOT.B	Dn	46	0 0ddd								
NOT.B	dadr	46	00ff ffff	[EXT]		[EXT]					
NOT.W	Dn	46	4 0ddd								
NOT.W	dadr	46	01ff ffff	[EXT]		[EXT]					
NOT.L	Dn	46	8 0ddd								
NOT.L	dadr	46	10ff ffff	[EXT]		[EXT]					
MOVE	Dn,SR	46	C 0sss								
MOVE	sadr,SR	46	11ee eeee	[EXT]		[EXT]					
MOVE	data16,SR	46	F C	yyyy							
NBCD	Dn	48	0 0ddd								
NBCD	dadr	48	00ff ffff	[EXT]		[EXT]					
SWAP	Dn	48	4 0rrr								
PEA	jadr	48	01jj jjjj	[EXT]		[EXT]					
EXT.W	Dn	48	8 0ddd								
MOVEM.W	reg−list,madr	48	10hh hhhh	mmmm		[EXT]		[EXT]			
MOVEM. W	reg−list,−(An)	48	A 0ddd	kkkk							
EXT.L	Dn	48	C 0ddd								
MOVEM. L	reg−list,madr	48	11hh hhhh	mmmm		[EXT]		[EXT]			
MOVE. L	reg−list,−(An)	48	E 0ddd	kkkk							
TST.B	Dn	4A	0 0rrr								
TST.B	dadr	4A	00ff ffff	[EXT]		[EXT]					
TST.W	Dn	4A	4 0rrr								
TST.W	dadr	4A	01ff ffff	[EXT]		[EXT]					
TST.L	Dn	4A	8 0rrr								
TST.L	dadr	4A	10ff ffff	[EXT]		[EXT]					
TAS	Dn	4A	C 0rrr								
TAS	dadr	4A	11ff ffff	[EXT]		[EXT]					
MOVEM. W	jadr,reg−list	4C	10jj jjjj	mmmm		[EXT]		[EXT]			
MOVEM.W	(An)+,reg−list	4C	A 0sss	mmmm							
MOVEM. L	(An)+,reg−list	4C	E 0sss	mmmm							
MOVEM. L	jadr,reg−list	4C	11jj jjjj	mmmm		[EXT]		[EXT]			
MOVEM. L	(An)+,reg−list	4C	E 0sss	mmmm							
TRAP	vector	4E	4 vvvv								

Tabella B.2 Codici oggetto delle istruzioni del microprocessore MC68000 in ordine numerico (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
LINK	An,d16	4E	5 0rrr	xxxx							
UNLK	An	4E	5 1rrr								
MOVE	An,USP	4E	6 0sss								
MOVE	USP,An	4E	6 1sss								
RESET		4E	7 0								
NOP		4E	7 1								
STOP	data16	4E	7 2	yyyy							
RTE		4E	7 3								
RTS		4E	7 5								
TRAPV		4E	7 6								
RTR		4E	7 7								
JSR	jadr	4E	10jj jjjj	[EXT]		[EXT]					
JMP	jadr	4E	11jj jjjj	[EXT]		[EXT]					
ADDQ.B	data3,Dn	5 bbb0	0 0ddd								
ADDQ.B	data3,dadr	5 bbb0	00ff ffff	[EXT]		[EXT]					
ADDQ.W	data3,Dn	5 bbb0	4 0ddd								
ADDQ.W	data3,An	5 bbb0	4 1ddd								
ADDQ.W	data3,dadr	5 bbb0	01ff ffff	[EXT]		[EXT]					
ADDQ.L	data3,Dn	5 bbb0	8 0ddd								
ADDQ.L	data3,An	5 bbb0	8 1ddd								
ADDQ.L	data3,dadr	5 bbb0	01ff ffff	[EXT]		[EXT]					
ST	Dn	50	C 0ddd								
DBT	Dn,label	50	C 1rrr	xxxx							
ST	dadr	50	11ff ffff	[EXT]		[EXT]					
SUBQ.B	data3,Dn	5 bbb1	0 0ddd								
SUBQ.B	data3,dadr	5 bbb1	00ff ffff	[EXT]		[EXT]					
SUBQ.W	data3,Dn	5 bbb1	4 0ddd								
SUBQ.W	data3,An	5 bbb1	4 1ddd								
SUBQ.W	data3,dadr	5 bbb1	01ff ffff	[EXT]		[EXT]					
SUBQ.L	data3,Dn	5 bbb1	8 0ddd								
SUBQ.L	data3,An	5 bbb1	8 1ddd								
SUBQ.L	data3,dadr	5 bbb1	01ff ffff	[EXT]		[EXT]					
SF	Dn	51	C 0ddd								
DBF	Dn,label	51	C 1rrr	xxxx							
SF	dadr	51	11ff ffff	[EXT]		[EXT]					
SHI	Dn	52	C 0ddd								
DBHI	Dn,label	52	C 1rrr	xxxx							
SHI	dadr	52	11ff ffff	[EXT]		[EXT]					
SLS	Dn	53	C 0ddd								
DBLS	Dn,label	53	C 1rrr	xxxx							
SLS	dadr	53	11ff ffff	[EXT]		[EXT]					
SCC	Dn	54	C 0ddd								
DBCC	Dn,label	54	D 1rrr	xxxx							
SCC	dadr	54	11ff ffff	[EXT]		[EXT]					
SCS	Dn	55	C 0ddd								
DBCS	Dn,label	55	C 1rrr	xxxx							
SCS	dadr	55	11ff ffff	[EXT]		[EXT]					
SNE	Dn	56	C 0ddd								
DBNE	Dn,label	56	C 1rrr	xxxx							
SNE	dadr	56	11ff ffff	[EXT]		[EXT]					
SEQ	Dn	57	C 0ddd								
DBEQ	Dn,label	57	C 1rrr	xxxx							
SEQ	dadr	57	11ff ffff	[EXT]		[EXT]					
SVC	Dn	58	C 0ddd								
DVC	Dn,label	58	C 1rrr	xxxx							
SVC	dadr	58	11ff ffff	[EXT]		[EXT]					
SVS	Dn	59	C 0ddd								
DVS	Dn,label	59	C 1rrr	xxxx							
SVS	dadr	59	11ff ffff	[EXT]		[EXT]					
SPL	Dn	5A	C 0ddd								
DBPL	Dn,label	5A	C 1rrr	xxxx							
SPL	dadr	5A	11ff ffff	[EXT]		[EXT]					
SMI	Dn	5B	C 0ddd								
DBMI	Dn,label	5B	C 1rrr	xxxx							
SMI	dadr	5B	11ff ffff	[EXT]		[EXT]					
SGE	Dn	5C	C 0ddd								
DBGE	Dn,label	5C	C 1rrr	xxxx							
SGE	dadr	5C	11ff ffff	[EXT]		[EXT]					
SLT	Dn	5D	C 0ddd								
DBLT	Dn,label	5D	C 1rrr	xxxx							
SLT	dadr	5D	11ff ffff	[EXT]		[EXT]					
SGT	Dn	5E	C 0ddd								
DBGT	Dn,label	5E	C 1rrr	xxxx							

Tabella B.2 Codici oggetto delle istruzioni del microprocessore MC68000 in ordine numerico (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
SGT	dadr	5E	11ff ffff	[EXT]		[EXT]					
SLE	Dn	5F	C 0ddd								
DBLE	Dn,label	5F	C 1rrr	xxxx							
SLE	dadr	5F	11ff ffff	[EXT]		[EXT]					
BRA	label	60	0 0	xxxx							
BRA	label	60	xx								
BSR	label	61	00	xxxx							
BSR	label	61	xx								
BHI	label	62	00	xxxx							
BHI	label	62	xx								
BLS	label	63	00	xxxx							
BLS	label	63	xx								
BCC	label	64	00	xxxx							
BCC	label	64	xx								
BCS	label	65	00	xxxx							
BCS	label	65	xx								
BNE	label	66	00	xxxx							
BNE	label	66	xx								
BEQ	label	67	00	xxxx							
BEQ	label	67	xx								
BVC	label	68	00	xxxx							
BVC	label	68	xx								
BVS	label	69	00	xxxx							
BVS	label	69	xx								
BPL	label	6A	00	xxxx							
BPL	label	6A	xx								
BMI	label	6B	00	xxxx							
BMI	label	6B	xx								
BGE	label	6C	00	xxxx							
BGE	label	6C	xx								
BLT	label	6D	00	xxxx							
BLT	label	6D	xx								
BGT	label	6E	00	xxxx							
BGT	label	6E	xx								
BLE	label	6F	00	xxxx							
BLE	label	6F	xx								
MOVEQ	data8,Dn	7ddd0	yy								
OR. B	sDn,dDn	8ddd0	0 0sss								
OR. B	sadr,Dn	8ddd0	00eeeeee	[EXT]		[EXT]					
OR. W	sDn,dDn	8ddd0	4 0sss								
OR.W	sadr,Dn	8ddd0	01eeeeee	[EXT]		[EXT]					
OR.L	sDn,dDn	8ddd0	8 0sss								
OR.L	sadr,Dn	8ddd0	10eeeeee	[EXT]		[EXT]					
DIVU'	sDn,dDn	8ddd0	C 0sss								
DIVU	sadr,Dn	8ddd0	11eeeeee	[EXT]		[EXT]					
DIVU	data16,Dn	8ddd0	F C	yyyy							
SBC	sDn,dDn	8ddd1	0 0sss								
SBCD	−(sAn),−(dAn)	8ddd1	0 1sss								
OR.B	Dn,dadr	8sss1	00ff ffff	[EXT]		[EXT]					
OR.W	Dn,dadr	8sss1	01ff ffff	[EXT]		[EXT]					
OR.L	Dn,dadr	8sss1	10ff ffff	[EXT]		[EXT]					
DIVS	sDn,dDn	8ddd1	C' 0sss								
DIVS	sadr,Dn	8ddd1	11ee eeee	[EXT]		[EXT]					
DIVS	data16,Dn	8ddd1	F C	yyyy							
SUB.B	sDn,dDn	9ddd0	0 0sss								
SUB.B	sadr,Dn	9ddd0	00ee eeee	[EXT]		[EXT]					
SUB.W	rs,Dn	9ddd0	4 tsss								
SUB.W	sadr,Dn	9ddd0	01ee eeee	[EXT]		[EXT]					
SUB.L	rs,Dn	9ddd0	8 tsss								
SUB.L	sadr,Dn	9ddd0	10eeeeee	[EXT]		[EXT]					
SUB.W	rs,An	9ddd0	C tsss								
SUB.W	sadr,An	9ddd0	11eeeeee	[EXT]		[EXT]					
SUB.W	data16,An	9ddd0	F C	yyyy							
SUBX.B	sDn,dDn	9ddd1	0 0sss								
SUBX.B	−(sAn),−(dAn)	9ddd1	0 1sss								
SUB.B	Dn,dadr	9sss1	00ffffff	[EXT]		[EXT]					
SUBX.W	sDn,dDn	9ddd1	4 0sss								
SUBX.W	−(sAn),−(dAn)	9ddd1	4 1sss								
SUB.W	Dn,dadr	9sss1	01ff ffff	[EXT]		[EXT]					
SUBX.L	sDn,dDn	9ddd1	8 0sss								
SUBX.L	−(sAn),−(dAn)	9ddd1	8 1sss								
SUB.L	Dn,dadr	9sss1	10ffffff	[EXT]		[EXT]					
SUB.L	rs,An	9ddd1	C tsss								

Tabella B.2 Codici oggetto delle istruzioni del microprocessore MC68000 in ordine numerico (continua)

Istruzione		Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
SUB.L	sadr,An	9 ddd1	11eeeeee	[EXT]		[EXT]					
SUB.L	data32,An	9 ddd1	F C	yyyy		zzzz					
CMP.B	sDn,dDn	B ddd0	0 0sss								
CMP.B	sadr,Dn	B ddd0	00eeeeee	[EXT]		[EXT]					
CMP.W	rs,Dn	B ddd0	4 tsss								
CMP.W	sadr,Dn	B ddd0	01eeeeee	[EXT]		[EXT]					
CMP.L	rs,Dn	B ddd0	8 tsss								
CMP.L	sadr,Dn	B ddd0	10eeeeee	[EXT]		[EXT]					
CMP.W	rs,An	B ddd0	C tsss								
CMP.W	sadr,An	B ddd0	11eeeeee	[EXT]		[EXT]					
CMP.W	data 16,An	B ddd0	F C	yyyy							
EOR.B	sDn,dDn	B sss1	0 0ddd								
CMPM.B	(sAn)+,(dAn)+	B ddd1	0 1sss								
EOR.B	Dn,dadr	B sss1	00ffffff	[EXT]		[EXT]					
EOR.W	sDn,dDn	B sss1	4 0ddd								
CMPM.W	(sAn)+,(dAn)+	B ddd1	4 1sss								
EOR.W	Dn,dadr	B sss1	01ffffff	[EXT]		[EXT]					
EOR.L	sDn,dDn	B sss1	8 0ddd								
CMPM.L	(sAn)+,(dAn)+	B ddd1	8 1sss								
EOR.L	Dn,dadr	B sss1	10ffffff	[EXT]		[EXT]					
CMP.L	rs,An	B ddd1	C tsss								
CMP.L	sadr,An	B ddd1	11eeeeee	[EXT]		[EXT]					
CMP.L	data32,An	B ddd1	F C	yyyy		zzzz					
AND.B	sDn,dDn	C ddd0	0 0sss								
AND.B	sadr,Dn	C ddd0	00eeeeee	[EXT]		[EXT]					
AND.W	sDn,dDn	C ddd0	4 0sss								
AND.W	sadr,Dn	C ddd0	01eeeeee	[EXT]		[EXT]					
AND.L	sDn,dDn	C ddd0	8 0sss								
AND.L	sadr,Dn	C ddd0	10eeeeee	[EXT]		[EXT]					
MULU	sDn,dDn	C ddd0	C 0sss								
MULU	sadr,Dn	C ddd0	11eeeeee	[EXT]		[EXT]					
MULU	data16,Dn	C ddd0	F C	yyyy							
ABCD	sDn,dDn	C ddd1	0 0sss								
ABCD	−(sAn),−(dAn)	C ddd1	0 1sss								
AND.B	Dn,dadr	C sss1	00ffffff	[EXT]		[EXT]					
EXG	Dn,Dn	C sss1	4 0ddd								
EXG	An,An	C sss1	4 1ddd								
AND.W	Dn,dadr	C sss1	01ffffff	[EXT]		[EXT]					
EXG	Dn,An or An,Dn	C sss1	8 1ddd								
AND.L	Dn,dadr	C sss1	10ffffff	[EXT]		[EXT]					
MULS	sDn,dDn	C ddd1	C 0sss								
MULS	sadr,Dn	C ddd1	11eeeeee	[EXT]		[EXT]					
MULS	data16,Dn	C ddd1	F C	yyyy							
ADD.B	sDn,dDn	D ddd0	0 0sss								
ADD.B	sadr,Dn	D dddC	00eeeeee	[EXT]		[EXT]					
ADD.W	rs,Dn	D ddd0	4 tsss								
ADD.W	sadr,Dn	D ddd0	01eeeeee	[EXT]		[EXT]					
ADD.L	rs,Dn	D ddd0	8 tsss								
ADD.L	sadr,Dn	D ddd0	10eeeeee	[EXT]		[EXT]					
ADD.W	rs,An	D ddd0	C tsss								
ADD.W	sadr,An	D ddd0	11eeeeee	[EXT]		[EXT]					
ADD.W	data16,An	D ddd0	F C	yyyy							
ADDX.B	sDn,dDn	D ddd1	0 0sss								
ADDX.B	−(sAn),−(dAn)	D ddd1	0 1sss								
ADD.B	Dn,dadr	D sss1	00ffffff	[EXT]		[EXT]					
ADDX.W	sDn,dDn	D ddd1	4 0sss								
ADDX.W	−(sAn),−(dAn)	D ddd1	4 1sss								
ADD.W	Dn,dadr	D sss1	01ffffff	[EXT]		[EXT]					
ADDX.L	sDn,dDn	D ddd1	8 0sss								
ADDX.L	−(sAn),−(dAn)	D ddd1	8 1sss								
ADD.L	Dn,dadr	D sss1	10ffffff	[EXT]		[EXT]					
ADD.L	rs,An	D ddd1	C tsss								
ADD.L	sadr,An	D ddd1	11eeeeee	[EXT]		[EXT]					
ADD.L	data32,An	D ddd1	F C	yyyy		zzzz					
ASR.B	count,Dn	E ccc0	0 0ddd								
LSR.B	count,Dn	E ccc0	0 1ddd								
ROXR.B	count,Dn	E ccc0	1 0ddd								
ROR.B	count,Dn	E ccc0	1 1ddd								
ASR.B	Dn,dDn	E rrr0	2 0ddd								
LSR.B	Dn,dDn	E rrr0	2 1ddd								
ROXR.B	Dn,dDn	E rrr0	3 0ddd								
ROR.B	Dn,dDn	E rrr0	3 1ddd								

Tabella B.2 Codici oggetto delle istruzioni del microprocessore MC68000 in ordine numerico (continua)

Istruzione	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
ASR.W count,Dn	E ccc0	4 0ddd								
LSR.W count,Dn	E ccc0	4 1ddd								
ROXR.W count,Dn	E ccc0	5 0ddd								
ROR.W count,Dn	E ccc0	5 1ddd								
ASR.W Dn,dDn	E rrr0	6 0ddd								
LSR.W Dn,dDn	E rrr0	6 1ddd								
ROXR.W Dn,dDn	E rrr0	7 0ddd								
ROR.W Dn,dDn	E rrr0	7 1ddd								
ASR.L count,Dn	E ccc0	8 0ddd								
LSR.L count,Dn	E ccc0	8 1ddd								
ROXR.L count,Dn	E ccc0	9 0ddd								
ROR.L count, Dn	E ccc0	9 1ddd								
ASR.L Dn,dDn	E rrr0	A 0ddd								
LSR.L Dn,dDn	E rrr0	A 1ddd								
ROXR.L Dn,dDn	E rrr0	B 0ddd								
ROR.L Dn,dDn	E rrr0	B 1ddd								
ASR dadr	E 0	11ffffff	[EXT]		[EXT]					
ASL.B count,Dn	E ccc1	0 0ddd								
LSL.B count,Dn	E ccc1	0 1ddd								
ROXL.B count,Dn	E ccc1	1 0ddd								
ROL.B count,Dn	E ccc1	1 1ddd								
ASL.B Dn,dDn	E rrr1	2 0ddd								
LSL.B Dn,dDn	E rrr1	2 1ddd								
ROXL.B Dn,dDn	E rrr1	3 0ddd								
ROL.B Dn,dDn	E rrr1	3 1ddd								
ASL.W count,Dn	E ccc1	4 0ddd								
LSL.W count,Dn	E ccc1	4 1ddd								
ROXL.W count,Dn	E ccc1	5 0ddd								
ROL.W count,Dn	E ccc1	5 1ddd								
ASL.W Dn,dDn	E rrr1	6 0ddd								
LSL.W Dn,dDn	E rrr1	6 1ddd								
ROXL.W Dn,dDn	E rrr1	7 0ddd								
ROL.W Dn,dDn	E rrr1	7 1ddd								
ASL.L count,Dn	E ccc1	8 0ddd								
LSL.L count,Dn	E ccc1	8 1ddd								
ROXL.L count,Dn	E ccc1	9 0ddd								
ROL.L count,Dn	E ccc1	9 1ddd								
ASL.L Dn,dDn	E rrr1	A 0ddd								
LSL.L Dn,dDn	E rrr1	A 1ddd								
ROXL.L Dn,dDn	E rrr1	B 0ddd								
ROL.L Dn,dDn	E rrr1	B 1ddd								
ASL dadr	E 1	11ffffff	[EXT]		[EXT]					
LSR dadr	E 2	11ffffff	[EXT]		[EXT]					
LSL dadr	E 3	11ffffff	[EXT]		[EXT]					
ROXR dadr	E 4	11ffffff	[EXT]		[EXT]					
ROXL dadr	E 5	11ffffff	[EXT]		[EXT]					
ROR dadr	E 6	11ffffff	[EXT]		[EXT]					
ROL dadr	E 7	11ffffff	[EXT]		[EXT]					

INDICE ANALITICO

A

B

C

D

E

F

H

I

J

L

M

N

O

P

R

S

T

U

V

X

Z

Il libro fornisce le informazioni indispensabili per conoscere a fondo l'architettura e il funzionamento del 68000 e ne illustra il linguaggio di programmazione relativo. Ogni singola istruzione Assembler 68000 viene descritta in modo ampio e dettagliato.
Nel testo sono contenuti molti esempi pratici di programmazione di cui viene fornita sia la versione in codice oggetto sia quella sorgente.
Contiene inoltre una sezione dedicata al processo di sviluppo del software in cui vengono spiegate tutte le fasi che vanno dalla definizione del problema alla manutenzione dei programmi, con particolare attenzione alla fase di progettazione del programma e a quella di debugging.
I numerosi esempi riportati trattano in modo esaustivo anche la programmazione dei dispositivi di input/output e i metodi di interfacciamento.
È perciò un libro indirizzato sia a chi non ha mai programmato in linguaggio Assembler sia a chi, già esperto programmatore, vuole conoscere le istruzioni del 68000 e le sue tecniche di programmazione.

L. 70.000 Cod. GY265 ISBN 88-7056-348-0

G. Kane D. Hawkins L. Leventhal

ASSEMBLER PER 68000